BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

MOBILE APPLICATION USING DEEP CONVOLUTIONAL NEURAL NETWORKS

MOBILNÍ APLIKACE VYUŽÍVAJÍCÍ HLUBOKÝCH KONVOLUČNÍCH NEURONOVÝCH SÍTÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. SEBASTIÁN POLIAK**
AUTOR PRÁCE

SUPERVISOR **Ing. JAKUB SOCHOR**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

This thesis describes a process of creating a mobile application using deep convolutional neural networks. The process starts with proposal of the main idea, followed by product and technical design, implementation and evaluation. The thesis also explores the technical background of image recognition, and chooses the most suitable options for the purpose of the application. These are object detection and multi-label classification, which are both implemented, evaluated and compared. The resulting application tries to bring value from both user and technical point of view.

Abstrakt

Táto práca popisuje proces tvorby mobilnej aplikácie, ktorá využíva hlboké konvolučné neurónové siete. Proces začína predstavením hlavnej myšlienky, po ktorej nasleduje produktový a technický návrh, implementácia a vyhodnotenie. Práca taktiež skúma technické pozadie rozpoznávania obrazu, a vyberá najvhodnejšie možnosti pre účely aplikácie. Tie sú detekcia objektov a multi-label klasifikácia, ktoré sú obe implementované, vyhodnotené a porovnané. Výsledná aplikácia sa snaží priniesť hodnotu z užívateľského aj technického hľadiska.

Keywords

photo-challenge mobile application, image recognition, convolutional neural network, object detection, multi-label classification, image dataset, client-server model, user testing

Klíčová slova

mobilná aplikácia s foto-výzvami, rozpoznávanie obrazu, konvolučná neurónová sieť, detekcia objektov, multi-label klasifikácia, dátová sada obrázkov, model klient-server, užívateľské testovanie

Reference

POLIAK, Sebastián. *Mobile Application Using Deep Convolutional Neural Networks.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jakub Sochor

Mobile Application Using Deep Convolutional Neural Networks

Declaration

Hereby I declare that this term project was prepared as an original author's work under the supervision of Ing. Jakub Sochor. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Sebastián Poliak
May 22, 2018

Acknowledgements

I would like to thank to my supervisor Ing. Jakub Sochor, for his guidance in this project, mainly in its core part - image recognition.
Additionally, I would like to thank to Ing. Rastislav Mrazik, for creation of the animated avatars, Mgr.Art. Juraj Poliak, for creation of the logo of the application, Bc. Zdeno Olšovský, for new ideas and discussions about the mobile part of the application, and to my family and friends, for overall support during this process.
Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Contents

Chapter 1

Introduction

We live in an era, where visual content is the main source of information exchange. Since a camera has become an inseparable part of a mobile phone, and therefore, always accessible, its usage for capturing everyday moments, usually followed by a share on a social media, has grown up enormously. From the observation of such pictures, it is possible to see that their content often includes quite ordinary things, for example a lunch or a dog.
This thesis aims to create a mobile application, which is based on the user behaviour of taking pictures of ordinary things. Additionally, it takes an advantage of recent development in the field of machine learning, and uses it to support the idea. The resulting application tries to bring value from both user and technical point of view.
The thesis describes the full development process of creating the application, starting with Chapter 2 Product Design, where the initial idea is proposed, together with user requirements and the current state of the art. Chapter 3 Technical Design describes the design of the application from the technical point of view, and provides the reasoning behind every decision. The background of a core part of the application - image recognition, is described separately in Chapter 4 Image Recognition Background, focusing on the options to built a suitable image recognition system. The implementation of every part, according to the proposed design, is described in Chapter 5 Implementation. At last, the implemented image recognition system, as well as the application itself, are evaluated in Chapter 6 Evaluation.

Chapter 2

Product Design

This chapter introduces the main idea and purpose of the application, the requirements for the user experience to be satisfactory, as well as the current state of the art.

2.1 Idea Proposal

The idea for the application originated from several different sources. Probably the most influential one, was a Slovak TV show called „VILOmeniny“ [37], where 2 families competed against each other, in various challenges. One of the challenges was to find 5 different items in their family house, within a time limit of 60 seconds. Combined with the current capabilities of machine learning, there was an idea to simulate this experience, using a mobile device with a camera.

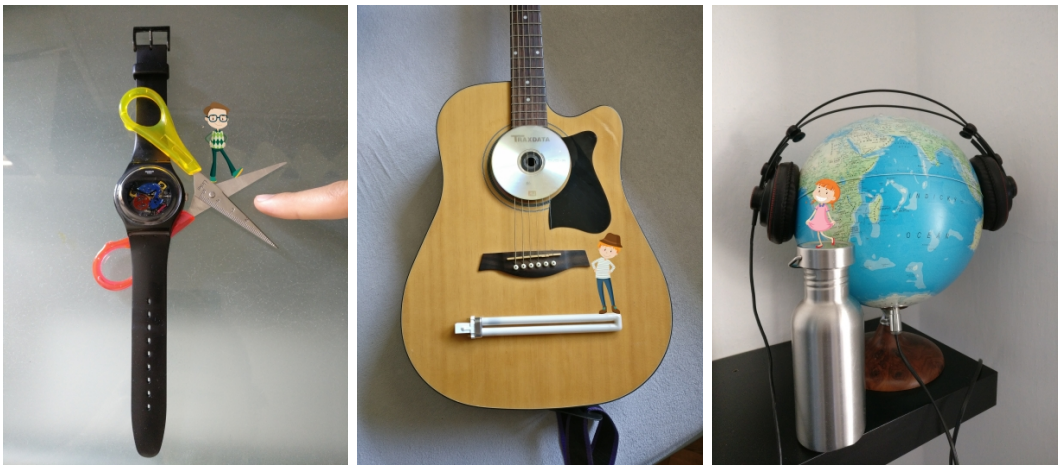


Figure 2.1: Example pictures used to create initial quest mockups.

The basic idea has been further changed, in order to better fit the user behaviour, and bring additional value. It starts by randomly generatating a list of items for each quest, with the objective to take a picture containing all the items. The quest is restricted by a time limit, and if completed in time, the user gets a new quest with a new list of items, that are more difficult to find in a real life. The idea also includes animated characters, which are placed inside the picture, allowing users to personalize, as well as making the pictures

more interesting, as illustrated in Figure 2.1. These and other features are described in the following Section 2.2. According to the description, the proposed product can be considered a photo-challenge mobile application.

2.2 User Requirements

To enhance the user experience, promote playfulness, and therefore, keep users interested for a longer time, the application needs to meet several requirements. Although it is difficult to estimate the influence on the final user experience without any feedback, these are the main requirements that have been considered:

- **Gameplay** - since the application can be described as a game, there needs to be a clearly stated goal. In this case, the goal is to take a picture containing given items. However, to keep users engaged after they have reached the goal, the game needs to be progressing. This is achieved by making the items in the list progressively more difficult to find in a real life. Additionally, a time limit is added to every quest, making the game more dynamic. If the time limit expires, a new list of items with similar difficulty can be generated, not giving users a possibility to really fail and stop playing.

- **Personalization** - allowing users to personalize within the application can help to develop their feeling for the game and make them return again. The main feature for this is the animated avatar, which accompanies the user throughout the game. The avatar could ideally be personalized and developed during the progression in the game, for example by changing clothes or adding gadgets, received as a reward for completing the quests. The resulting avatar is placed in a certain position, into every taken picture. There is an infinite amount of ways, how the items in the picture can be arranged, how the picture can be taken, and how the animated avatar can be placed inside the picture. The goal is to motivate users to create pictures, that work together with the animated avatar and are potentially artistic, similarly to the pictures shown in Figure 2.1.

- **Community** - having an active community can improve the overall user experience. Allowing users to share their pictures on other social media could bring new users, as well as promote the application.
  The users would probably appreciate, if they could see the pictures of others, and interact directly in the game. This could be achieved by allowing them to visit the user profiles, showing their pictures and the current progress. Eventually, the pictures could get rated, or reported if needed. Another idea is to implement features such as a picture of the day, where a daily picture is chosen, and shown with the corresponding user name. These features, however, have not been developed in the scope of this thesis.

- **Efficient onboarding** - the attention span of users is limited, usually giving the application only one try. The onboarding and learning process should therefore feel easy and natural, making the users interested as soon as possible. However, for the reasons described in Chapter 3, an authentication will be required. This can slow down the onboarding process, and needs to be done with the consideration of user experience.

- **User interface** - the user interface has been designed in a way, to quickly understand the idea and functionality of the application, without need to include any additional tutorials. The main part of the application is composed of a tab view with two basic tabs, which are Gallery and Quest, shown in Figure 2.2. Gallery view contains the pictures of all completed quests, as well as the basic information about the current progress. On click, the pictures can be viewed in detail and shared. Quest view guides the users through the whole process of completing quests. Whenever a picture is taken, the items get evaluated, showing a check or cross sign for every item in the list. If all the items checks the quest is completed, and an avatar can be chosen, moved around and placed into the picture, which is the last part of the view. On top of both views, there is a drawer navigation, where sections such as About or Settings could be placed.

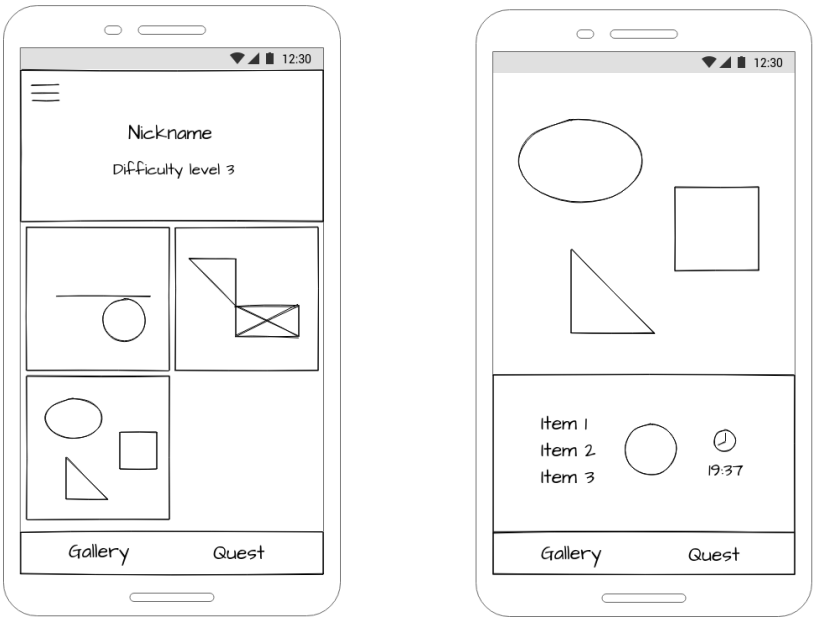


Figure 2.2: Wireframes of the user interface, showing an outline of Gallery and Quest views. The objects in the picture are represented as geometric shapes.

- **Functionality** - as described in Chapter 3, the application contains a soft computing part, the image recognition, which is not capable of accurately performing all the time. A possibility of an occasional failure needs to be considered, in connection with the user experience, and it's parameters need to be set accordingly. Additionally, the image recognition needs to be done in a reasonable time, so the users do not get frustrated while waiting. All these requirements have been considered in Chapter 5.

2.3 State of the Art

The motivation and reasoning behind why the proposed application could potentially become successful, comes from the current state of art and the user behaviour in general. Looking at the social media applications such as Instagram [15], it is possible to see that users like to take pictures of ordinary things, such as food, books or animals. Another

example of similar user behaviour, mainly in the usage of camera and real life scene, is found in the application Pokémon Go [25]. In this application, the users are meant to catch the pokémon, compared to finding items, in case of the proposed application. In 2016, Pokémon Go received The Game Award for Best Mobile/Handheld Game. An example of a real Instagram profile, as well as a real life scene in Pokémon Go, is shown in Figure 2.3.

At the time of development of the proposed application, there were not any known mobile applications on Google Play [12] nor App Store [2], that would have the same format. There existed some photo-challenge applications, for example an application called lifeshot [19], which challenges users to take a picture at the same time and share it, however, none of them used image recognition of any kind.
The applications, that were using image recognition, are for example the application Not Hotdog [23], which is able to classify one object in the picture, or the application Mushroom Identify [22], which is able to classify the type of mushroom in the picture. None of these related to the previously mentioned game format, however, they proved the concept of using camera and image recognition in a mobile application.

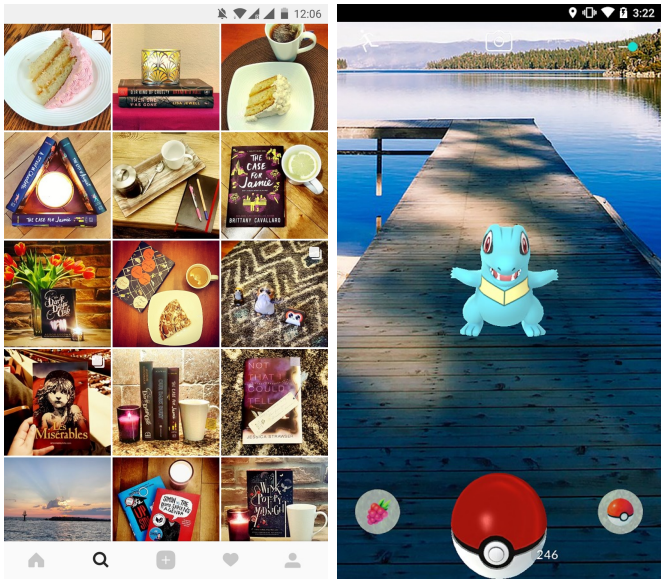


Figure 2.3: On the left, an example of a real Instagram profile, with the pictures of ordinary things. On the right, a screenshot from Pokémon Go, showing the usage of a real life scene.

2.3.1 Emoji Scavenger Hunt

An application with the similar idea has been published, a few days after the release of the proposed application. It has been introduced by Google, with the release of TensorFlow.js [41], which is a browser based JavaScript library, that can be used for training and deployment of machine learning models. The application is not available on Google Play or App Store, for the reason of being more like a demonstration of the library running in the browser, and its source code has been made publicly available. The application is called Emoji Scavenger Hunt [39], with the idea of taking a picture of the object that is shown on the screen as an emoji.
However, there are several main differences compared to the application developed in this

project, both in the concept and user experience. The challenges always contain a single item, that is needed to be found within a very short time limit, without any progression. For the demonstration purposes, the application also tries to say any item, it is currently able to recognize in the picture. On the other hand, the challenges in the application developed in this project contain several items, with the goal of making the user arrange the items in an artistic way, personalize using animated avatars, and create a picture that can be further used. It also aims to create a platform, where the pictures are stored, and give users a sense of progression, by being able to access their current state at any time.
Additionally, there is a difference from the technical point of view. Recognizing only one object in the image is a single-label classification problem, described in Section 4.4, compared to a multi-label classification problem, solved in this project.

Figure 2.4: A logo of Emoji Scavanger Hunt.

Chapter 3

Technical Design

This chapter describes all the components, necessary to build the desired application, as well as the selected architecture, and the reasoning behind it.

3.1 Main Components

The functionality of the application can be divided into four fundamental components, which are mostly independent, have different requirements, and are responsible for different roles. In the end, the components are arranged together according to the pipeline in Figure 3.1, which is responsive to the user interaction. The components can be listed as:

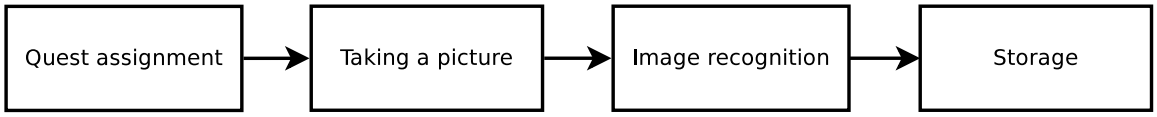


Figure 3.1: A pipeline of the functional components

- **Quest assignment** - the output of this component is a list of items, which are meant to be in the picture, and are recognized after the picture has been taken. The component leads the user through the whole progress in the game, assigning the quests at every difficulty level. Therefore, the items need to be divided into difficulty categories, and possibly context categories - fruits, animals, cutlery etc., in order to have more diverse quests. According to these requirements, a random list is generated. The list is generated again, in case the user fails to complete the given quest in time. In this case, there is another requirement, because we do not want to generate the same list as before.
- **Taking picture** - the application needs to use a camera, which is the only way to complete a quest. The picture has to be taken directly from the application, meaning that it cannot be uploaded externally from a gallery. The reasoning behind this is to have a full control over user experience, as well as making sure that the picture is taken by the actual user. Additionally, this way the properties of the camera and the resulting picture (eg. size or ratio) can be adjusted, and better fit the overall design.
- **Image recognition** - the core functionality, which evaluates the taken picture - determines, whether it contains all the assigned items. Technically, it can be formulated

as an image recognition problem, for which an entire Chapter 4 Image Recognition Background is devoted. A proper solution should ideally have an accuracy close to human-like level. It should be computationally feasible, and work under different constrains, such as time or operation cost, so it can be integrated with the rest of the application.

- **Storage** - it is required to store several kinds of data, such as user information, pictures, actual progress or user settings. While some of them might be stored on the device itself, the database would bring and advantage of accessing the same state in the application, from different devices.

3.2 Client-Server Model

Considering the previously listed roles and requirements, it has been decided to build the application using a client-server model. The client-server model is a distributed application structure that partitions task or workloads between the providers of a resource or service, called servers, and service requesters, called clients [5].
The client, in this case is the application running on a mobile device, which is responsible for taking a picture, communication with the server, and user interaction through the user interface, together with presentation of received data. The image recognition, as well as the functionality that generates and assigns quests to the users, are handled by the server. Another part of the server is the database. A division of the roles between client and server is illustrated in Figure 3.2

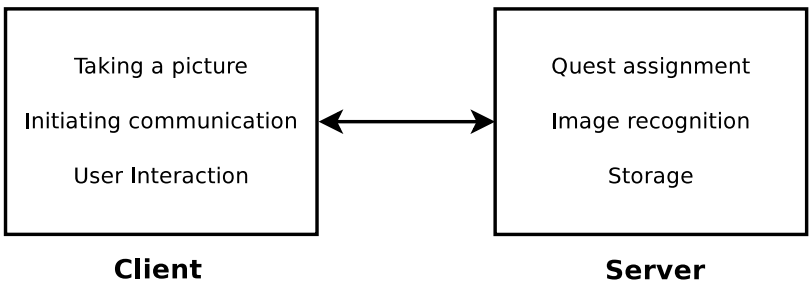


Figure 3.2: A division of roles between client and server.

There would be an option to have all the functionality contained directly in the mobile application, which is technically feasible. However, the choice of the client-server model has been made based on the advantages it brings, which are:

- **Flexibility** - the game logic can be easily modified and deployed during production. The changes on the server are instantly distributed to the clients, without any need to update the application. This offers an opportunity to quickly create new quests, for example during seasonal events such as Christmas or Easter. Another advantage is that the image recognition model can be improved, and switched anytime without user's notice.

- **Performance** - all the computation is done on the server, and therefore, the client's device needs to use very little computational power. This results in longer battery

life, because, mainly the image recognition part could drain a lot of battery. The system requirements for the device also get lower.

- **Feedback** - the images, that are sent to the server for image recognition can be stored, analyzed and used as a new input to further improve the image recognition model.
- **Platform independence** - the same server can be used across all different mobile platforms.

These disadvantages have been also considered:

- **Network dependence** - the internet connection is needed in order to communicate with the server. This could be problematic if the quest items are set somewhere outside, and it would be solved if the game logic was present in the device itself and the application could run offline. Nowadays, however, most of the users have a data connection from their telecommunication provider. Another possibility would be to allow users taking a picture offline, and submitting it for the evaluation later on, when the connection is available.
- **Operation cost** - running the computations on the server can get costly, especially if a GPU is used, which would be an ideal scenario for the inference of image recognition.
- **Maintenance and scaling** - the game logic of all clients is centralized. The number of clients can grow over time, and the server has to be able to handle multiple incoming requests at once, be safe, reliable and always accessible.

3.2.1 Authentication

With the choice of client-server model, another possibility is emerging. This is to implement user authentication in the application. It can be understood as a role of both client and server. The reasoning behind this choice might be

- **User-generated content stored in the database** - users are identified by a unique ID, according to which their content can be stored in the database. This way, the users can access their content from any device, they log in. Additionally, the content is not stored on the device, which mainly in case of images, might save the device's space.
- **Community** - the storage of user-generated content allows the option of creating a community, described in Section 2.2. This is possible due to having a way of accessing the content of other users.
- **Feedback about user state** - the authentication allows having even more feedback, than mentioned earlier. This is possible by having an information about the current state of the user, stored in the database. In the context of this application, the current state of the user corresponds to the difficulty level, the actual quest, and all the completed quests. The feedback could be used to create certain statistics, and improve the user experience.

Using authentication also brings disadvantages, mainly from the user point of view

- **Slower onboarding process** - by implementing authentication, a user registration would be needed the first user starts the application, which slows down the overall onboarding process. It could be partly solved by using federated identity providers such as Facebook [38], Google [40] or Twitter [48] for the purpose of authentication, and therefore, the users can log in directly without any registration.
- **Untrusting users** - the users might not feel safe providing personal information for the application. The only needed information would probably be an email, and the usage of the user information would be specified in the Privacy Policy.

Chapter 4

Image Recognition Background

This chapter describes the technical background of the core part of the application - image recognition, necessary for evaluation of the quest images. It focuses on current options to built a suitable image recognition system, according to the requirements of this project.

4.1 Overview

Image recognition is a task of determining whether or not the image data contains a specific object, feature, or activity. It belongs to the field of Computer Vision [6], along with other tasks such as motion analysis, scene reconstruction, or image restoration. There are several different varieties of the recognition problem:

- **Classification** - a problem, where one or more object classes are recognized in the image. An example might be an animal classifier, recognizing the species of the animal in the image.
- **Identification** - can be described as recognizing an individual instance of the object, for example identification of handwritten letters, a specific person's face or fingerprint.
- **Detection** - finding of smaller regions of interest in the image, which can be further analyzed to produce a correct interpretation. A few examples are detection of possible abnormal cells or tissues in medical images or detection of a vehicle in an automatic road toll system.

According to the listed problems, classification seems to best suite the problem of determining, whether the quest image contains certain items - object classes. More precisely, it is addressed as a multi-label classification problem, described in Section 4.4. However, the problem of detection, particularly object detection, has also been considered and is described in Section 4.5.
The techniques used for solving the problem of image recognition have developed over time, and might differ according to the given problem. A complete image recognition system is usually built out of several parts, such as image pre-processing, feature extraction and a machine learning model, while all these parts have an effect on the final system performance. The machine learning models such as Artificial Neural Network (ANN) [21], Support Vector Machines (SVM) [33] or Random Forest [27], have all been applied to solve the problem. These models vary in their accuracy, and while some of them might still be used in certain scenarios, the model that currently seems to work the best is Convolutional Neural Network (CNN).

4.2 Convolutional Neural Network

A convolutional neural network (CNN, or ConvNet) [8] is a class of deep, feed-forward artificial neural networks, that has been successfully applied to analysing visual imagery. The example tasks it has been applied to might be identifying faces, objects or traffic signs, as well as powering vision in robots and self-driving cars. In general, the architecture of a convolutional neural network can be divided into 2 parts - **feature extraction from image** and **classification** (see Figure 4.1).

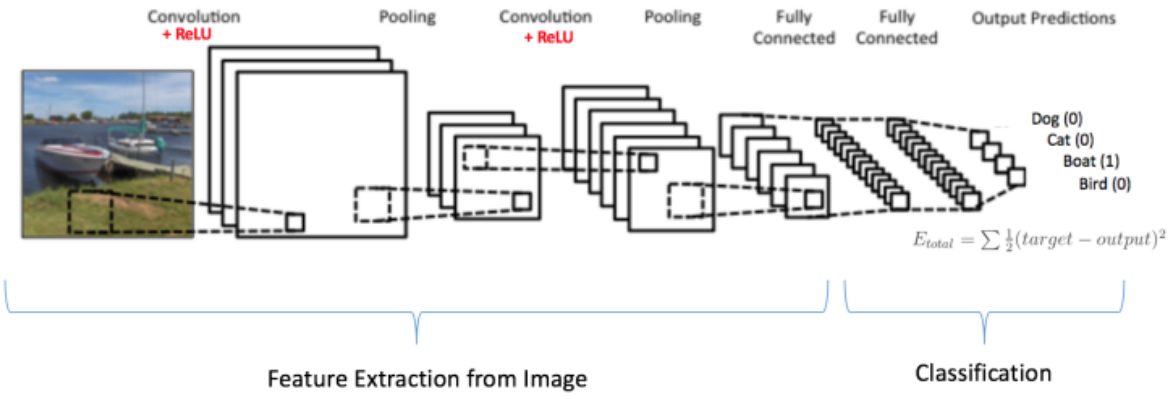


Figure 4.1: Structure of a simple convolutional neural network, taken from [4])

The feature extraction part of a convolutional neural network consists of these building blocks:

- **Convolution** - is a mathematical operation on two functions producing a third function, that is typically viewed as a modified version of one of the original functions [7]. The convolutional neural network has got its name according to this operation, and its purpose in this case is to extract features from the input image. The input image can be basically considered a matrix of pixel values. Another matrix called a 'filter' or 'kernel' or 'feature detector' is used and the convolution is done by sliding the filter over the image and computing the dot product. The resulting matrix is often called the 'Activation Map' or the 'Feature Map'. The numeric values of the filter matrix determine the performed operation, which could be for example edge detection, sharpen or blur, resulting in a different feature map being extracted.

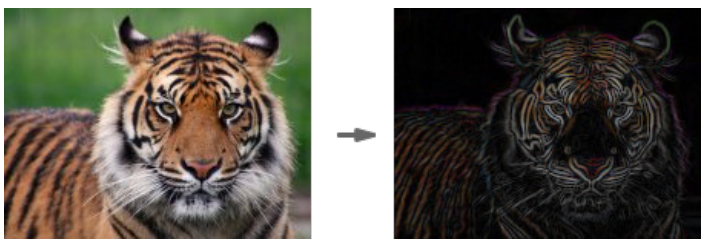


Figure 4.2: An example of usage of convolution for the edge detection, taken from [20]

- **Rectified Linear Unit** - known as ReLU, is a non-linear operation, written as $y = max(0, x)$. This operation is a part of a convolutional layer in the convolutional neural

network. It is applied after every convolution operation to every pixel in the feature map, replacing all negative pixel values by zero. The reasoning behind using this operation, is to introduce non-linearity to the network, since all the other operations are linear. Other functions such as sigmoid (Equation 4.2) or hyperbolic tangent ($tanh\ x = (e^x - e^{-x})/(e^x + e^{-x})$), most known from the regular neural networks could be also used. However, in case of a convolutional neural network, ReLU has been found to perform the best. A plot of this operation is shown in Figure 4.3.

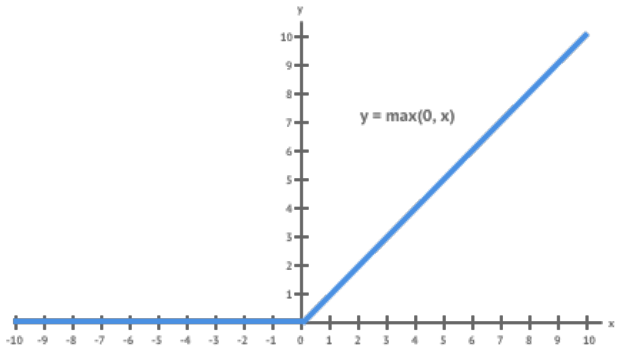


Figure 4.3: A plot of ReLU operation, taken from [20].

- **Pooling** - it is common to insert a pooling layer, after every convolutional layer in the structure a convolutional neural network. Its function is to progressively reduce the spatial size of the representation. This reduces the amount of parameters and computation in the network, and also helps to control overfitting. Most commonly, a pooling layer is using *max* operation with filters of size 2x2, which downsample every depth slice in the input by 2 along both width and height, discarding 75% of the activations. An example is shown in Figure 4.4).

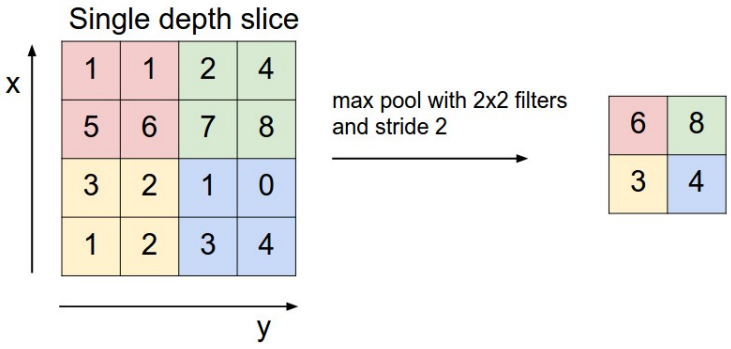


Figure 4.4: An example of max pooling, downsampling the input by taking a maximum of 4 numbers, taken from [8]

The classification part consists of fully connected layers, which outlines the architecture of the regular feedforward artificial neural network.

4.3 Image Datasets

A dataset is a collection of data, which is used to train a machine-learning model. In case of supervised learning [32], each data instance is a pair of an input object and its corresponding output value. The data is usually split into train, validation, and test subsets, used for different purposes.

- **Training set** - a set of examples used to fit the parameters of the model.
- **Validation set** - provides an unbiased evaluation of a model fit on the training dataset, in order to tune the model's parameters.
- **Test set** - provides an unbiased evaluation of a final model fit on the training dataset.

To train an image recognition model, the input data is a set of images, while the output contains object classes found in the image, and possibly an information about object locations. This information is usually provided in a form of bounding box of a rectangle shape, or possibly using a superpixel segmentation (see Figure 4.5).
Several image datasets are publicly available, and vary in the context of their images. In case of this project, the context is the ordinary objects, that can be found in the real life. The datasets fitting this description, that have been considered in this project are COCO: Common Objects in Context [47], The Open Images dataset [42], and ImageNet [45].

Figure 4.5: Example instances from COCO: Common Objects in Context on the left, using the superpixel segmentation, and The Open Images on the right, using the bounding boxes.

COCO: Common Objects in Context is a large-scale object detection, segmentation, and captioning dataset. It contains 330 000 images, from which more than 200 000 are labeled. There are total of 1.5 million object instances in 80 object categories. The objects in images are segmented, using a bounding box and a superpixel segmantation.
An even larger-scale dataset is The Open Images. All together, it contains more than 9 million images and thousands of classes, from which it is claimed that around 5000 are trainable.
ImageNet is an image database organized according to a word hierarchy. Currently, the nodes in the hierarchy contain five hundred images on average.
A subset of object classes needs to be selected from the chosen dataset. The selection should include only the object classes suitable for the purpose of the application, maximizing the potential of the trained image recognition model. There would also be a possibility of joining

multiple datasets, which could be done by making a selection from their overlapping object classes. This would potentially create a larger and more diverse dataset, and could result in a higher accuracy of the trained image recognition model

4.4 Multi-label Classification

This section focuses on solving the image recognition as a classification problem, with the focus on multi-label classification. In general, a classification problem can be divided into three variants, which are:

- **Binary classification** - a task of classifying the instances into one of two classes.
- **Multiclass classification** - a task of classifying instances into precisely one of more than two classes. The task still solves a single-label problem.
- **Multi-label classification** - a task of assigning multiple labels to each instance.

In context of this project, multi-label classification has been used for the reason of classifying multiple objects in the quest image.
The approaches to create a multi-label classifier might differ, depending on the used machine learning model. While some models can be directly modified to provide a multi-label output, others need a method of transforming the problem. One of such methods is called binary relevance [43], which divides the task into multiple binary tasks. This way, an independent binary classifier is trained for each label, and used to predict the label on every instance.
To use a neural network as a multi-label classifier, its output layer can be modified. A binary neural network classifier usually contains only one neuron in the output layer. To generalize the neural network for multiclass classification, the output layer consists of the number of neurons, corresponding to the number of classes. In this case, the output layer usually uses a softmax function (Equation 4.1). To achieve a multi-label classification, the activation function is changed, usually to sigmoid function (Equation 4.2). This is the approach, that has been used in this project. Although the comparison with binary relevance method would be interesting to see, it has not been explored. The reasoning behind this is that having a high number of selected classes would result in high operation, when training and inferring separate neural networks.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for j=1,...,K} \tag{4.1}$$

$$\sigma(z) = \frac{e^x}{e^x + 1} \tag{4.2}$$

A training of a multi-label image classifier does not require having the input images with information about object locations, compared to object detectors. The only needed information is the output of object classes. The input images, however, should contain multiple object classes, which has been explored in Section 5.1.1, where the trained multi-label classifier failed to classify multiple objects in a single instance.

4.5 Object Detection

The second approach that has been used in this project, is solving the image recognition part using object detection, The added value of this approach is, that along with the object classes, the output provides an information about object locations in the image. Although this information is not needed for the purpose of the application, the algorithm uses different methods of evaluating the image, which might result in a different accuracy. The additional information about the object location can always be ignored, using the object detector as a classifier.
Object detection, or multi-class object detection, in case multiple kinds of objects are detected in a single image, can be formulated as a classification problem where the windows of different sizes are taken from an input image at all possible locations, and proposed to an image classifier. However in reality, with respect to computational complexity, the windowing needs to be done in a clever way, and not really taking into consideration all the possible locations and window sizes. Such an optimization gets complicated by the fact, that objects can have different sizes and aspect ratios. The location of the object is usually represented as a bounding box, already mentioned in Section 4.3. This applies to both, the input training images, which need to have this information, as well as the output provided by model inference.
There are several architectures used for object detection, which are based on different principles, and vary in their speed and accuracy. The ones that have been considered are Single Shot MultiBox Detector [49] and Faster R-CNN [44].

4.5.1 Single Shot MultiBox Detector

This section is based on the paper SSD: Single Shot MultiBox Detector [49].

Single Shot MultiBox Detector (SSD) is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes. This is followed by a non-maximum suppression step to produce the final detections.

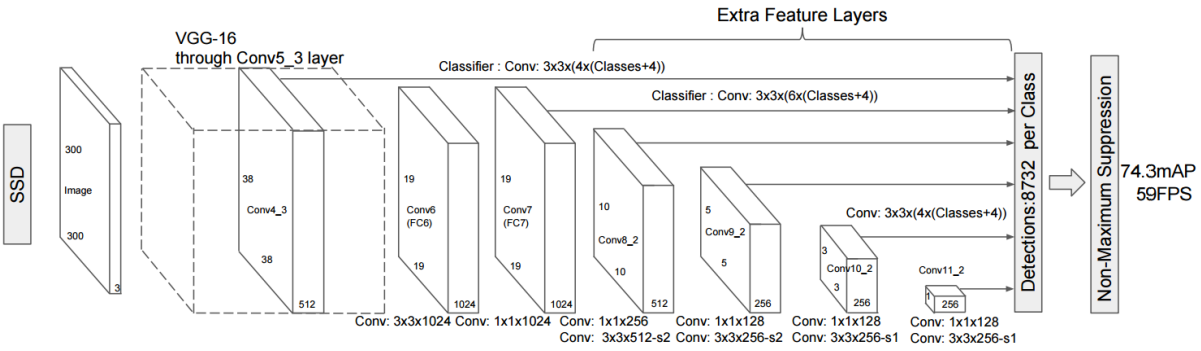
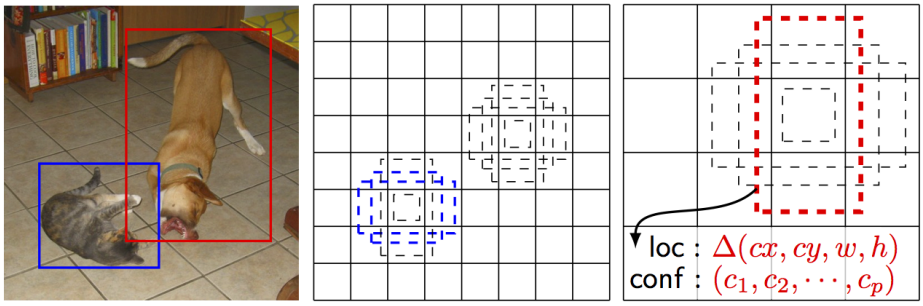


Figure 4.6: Structure of Single Shot MultiBox Detector, taken from [49].

An input to the network is an image of a fixed size, either 300x300 or 512x512 pixels. The structure of Single Shot MultiBox Detector is composed of different types of layers,

shown in Figure 4.6. The early network layers are based on a standard architecture used for high quality image classification. The approach presented in the paper, calls this part the base network, and uses the VGG-16 network, claiming that other networks should produce a similarly good results. The remaining structure of the network is added in a way to produce detections, using key features such as:

- **Multi-scale feature maps for detection** - convolutional feature layers are added to the end of the base network. These layers decrease in size progressively and allow predictions of detections at multiple scales. The convolutional model for predicting detections is different for each feature layer.
- **Convolutional predictors for detection** - each layer can produce a fixed set of detection predictions using a set of convolutional filters. These are indicated on top of the SSD network architecture (see Figure 4.6).
- **Default boxes and aspect ratios** - at the top of the network, the SSD evaluates a small set of default boxes of different aspect ratios, at each location in several feature maps with different scales (see Figure 4.7). For each default box, the shape offsets and the confidences for all object categories ((c1, c2, · · · , cp)) are predicted.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 4.7: Example of matching default boxes to the ground truth boxes, taken from [49].

The approach of Single Shot MultiBox Detector is more light-weight compared to others, which results in a significant improvement in speed, reaching 59 frames per second (FPS). It has often been used for a real-time object detection, for example in a video footage, which could even be performed on a mobile device.

4.5.2 Faster R-CNN

This section is based on the paper Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks [44].

Another approach called Faster R-CNN, is driven by the success of region proposal methods and region-based convolutional neural networks (R-CNNs). It builds on the previous model called Fast R-CNN, which achieves near real-time rates using deep networks, when ignoring the time spent on region proposals.
Faster R-CNN is composed of two modules. The first module is a deep fully convolutional network, known as a Region Proposal Network (RPN), and the second module is the Fast

R-CNN detector, that uses the proposed regions to detect object classes. The goal of Faster R-CNN is to share the computation between the networks of both modules, by having a common set of convolutional layers. The entire system is a single, unified network used for object detection, shown in Figure 4.9.
A Region Proposal Network takes an image of any size as an input, and outputs a set of rectangular object proposals, each with an objectness score. Objectness score measures a membership of the region to a set of object classes compared to background. To generate region proposals, a window is slided over the convolutional feature map output of the last shared convolutional layer. The features from the window are used as an input to the box-regression and box-classification layers. At each sliding-window location, multiple region proposals are simultaneusly predicted, where the number of maximum possible proposals is denoted as k. The box-regression layer has $4k$ outputs of the coordinates of k proposals, and the box-classification layer outputs $2k$ scores that estimate probability of object for each proposal. The k proposals are parameterized according to k reference boxes, called anchors. The functionality of Region Proposal Network is illustrated in Figure 4.8.

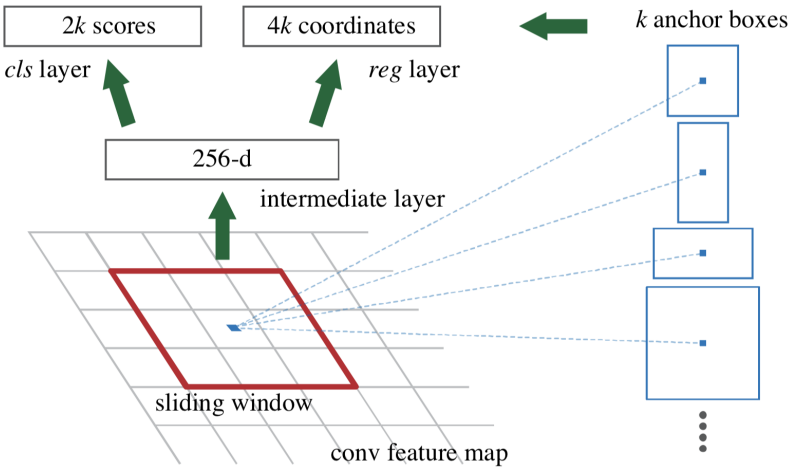


Figure 4.8: A functionality of Region Proposal Network, taken from [44]. Box-regression and box-classification layers are represented as *reg* and *cls* layers.

As mentioned, a technique that allows sharing the convolutional layers between the two networks, rather than training the networks separately, has been developed. The technique is based on three ways of training:

- **Alternating training** - Region Proposal Network is trained at first, and the proposals are used to train Fast R-CNN. The tuned convolutional layers are then used to initialize Region Proposal Network again, and this process is iterated.
- **Approximate joint training** - Region Proposal and Fast R-CNN networks are merged into one network during training. The forward pass normally generates precomputed proposals for Fast R-CNN detector. For the backward propagation, the loss of both Region Proposal and Fast R-CNN networks is combined for the shared convolutional layers. The backward propagation omits the gradients with respect to the box coordinates, which is why the approach is called approximate.

- **Non-approximate joint training** - the backward propagation in this approach also involves gradients with respect to the box coordinates, which was ignored in the approximate joint training. The reasoning behind this is that RoI (Region of Interest) layer in Fast R-CNN accepts the convolutional features, as well as the predicted bounding boxes as an input.

Sharing the convolutional layers makes the region proposal step nearly cost-free, which allows the entire network to operate at 5 frames per second on GPU.

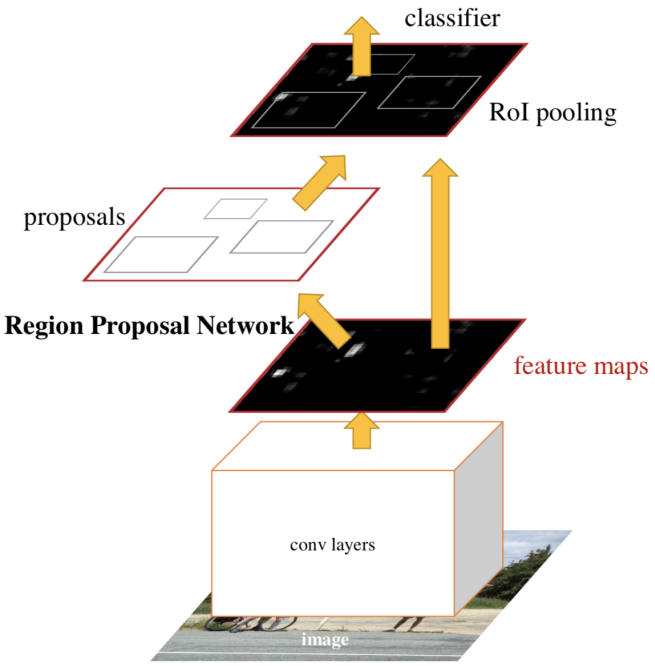


Figure 4.9: Faster R-CNN as a unified network used for object detection, taken from [44].

Chapter 5

Implementation

This chapter is devoted to the implementation part of the application, starting with the description of process of training image recognition models. The rest of the chapter is divided into two main sections, outlining the client-server model proposed in Chapter 3.

5.1 Training Image Recognition Models

Both multi-label classification, as well as object detection models, have been trained and are described in this section. The models have been trained on the image datasets described in Section 4.3. The preparation of the datasets to the correct input format has been done using custom scripts, in case of The Open Images, and using provided COCO API in case of Common Objects in Context. The training has been done on GPU, using infrastructure of MetaCentrum, mentioned in Acknowledgements.
The actual usage of the suitable model is described in Section 5.3.3. A brief description of training accuracy is provided in this section, however the models are properly evaluated in Section 6.1.

5.1.1 Multi-label Classifier

A multi-label image classifier has been built based on Inception V3 model [46], using Keras library. The model has been initialized with ImageNet weights and modified for multi-label classification, according to Section 4.4. The modification consisted of setting the parameter `include_top=False`, and adding a custom output layer. This layer contains the number of neurons, corresponding to the number of classes, with sigmoid activation function. The other parameters of the model were set to `loss='binary_crossentropy'` and `optimizer='adam'`.
An image data generator has been created, yielding the batches of size 128. The generator preprocesses the images and rescales them to target size of 299 per 299 pixels. Additionally, it performs a data augumentation, which randomly flips the images in horizontal direction. The model has been trained for 30 epochs, each containing a number of steps calculated as `steps_per_epoch=samples/batch_size`. A validation accuracy has been evaluated after every epoch, saving the model checkpoint with the highest score.
Initially, the model had been trained on The Open Images dataset, from which a subset of 20 classes, according to the desired gameplay of the application was selected. Although, the validation accuracy of the best model checkpoint reached 0.9173, it failed to properly classify multiple objects in an instance. This was caused by the fact, that the dataset rarely

contains image instances, which contain multiple objects from the selected subset at once. The result was, that the model was able to classify only a single object in the image. Considering this issue, a different dataset had to be used. COCO: Common Objects in Context dataset was more promising in this case, because its images often contain 2-5 of the selected objects at once. A subset of 29 objects from this dataset was selected in a similar way. The model has been trained with the same parameters, and the best model checkpoint reached validation accuracy of 0.9641. This model actually ended up being used in the final version of the application.

5.1.2 Object Detector

Both Faster R-CNN and Single Shot MultiBox Detector arhitectures presented in Section 4.5 have been trained as the object detection models. The training has been done using Tensorflow Object Detection API [35]. The initial model checkpoints that have been used are `ssd_inception_v2_coco` and `faster_rcnn_inception_v2_coco`. The same subset of 29 object classes from COCO: Common Objects in Context has been selected. Since it is the object detection, the input images need to contain the object classes with the corresponding bounding boxes. Using Tensorflow Object Detection API, the input needs to be in TFRecord file format [14], which has been created using COCO API and the attached scripts, resulting in `train.record` and `eval.record` files. Additionally, `label_map.pbtxt` file containing the mapping of object classes to neurons in the output layer needs to be provided.

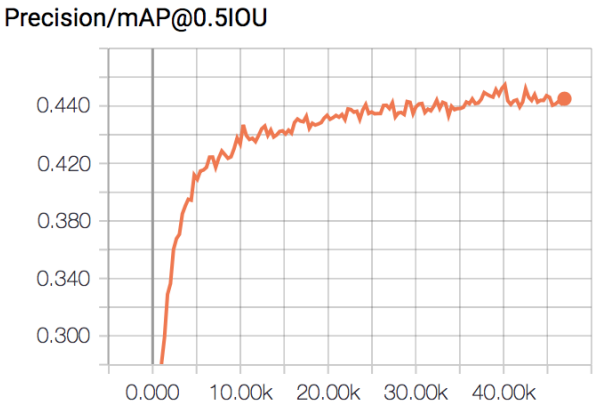


Figure 5.1: A training curve of Single Shot MultiBox Detector, after 47 thousands of steps.

The initial learning rate for the training of Single Shot MultiBox Detector has been set to 0.001, with decay factor of 0.95. For Faster R-CNN, the initial learning has been left to a default value of 0.0003, decreasing by a factor at 1000th and 15000th step. The batch size was set to 32 for both training configurations. During training, the model has been simultaneously evaluated and inspected using TensorBoard [34]. Single Shot MultiBox Detector model has been trained for 47 thousands of steps, and reached mean average precision (mAP) of 0.45@0.5IOU, further explained in Section 6.1. A training curve of Single Shot MultiBox Detector can be seen in Figure 5.1. Faster R-CNN model has been trained for 54 thousands of steps, and reached mean average precision of 0.54@0.5IOU. The

trained models have been exported for the inference as frozen graphs, with the parameters `iou_thresholds:0.0`, referencing the Intersection Over Union described in Section 6.1.1, and `score_thresholds:0.0`, which has been done manually.

5.2 Client Side

As mentioned in Chapter 3, the client side represents the application running on a mobile device. It is responsible for taking a picture, sending the requests to the server, and presenting the received data through user interface. Additionally, a role of authentication from the client's point of view, is described in this section.
Most commonly, the mobile applications are developed for two main mobile platforms - Android [24] and iOS [16]. The process of development is usually done through the development frameworks, provided directly by these platforms, resulting in the applications, compatible only with the specific platform. Android is an open source, Linux-based software stack, created for a wide array of devices and form factors. The development for Android platform is done using Java programming language, or more recently using Kotlin. The iOS is mobile operating system running in Apple devices, which creates an independent platform from Android. The development for this platform is done using Objective-C, or more recently Swift programming language.

5.2.1 React Native and Redux

Additional approaches for developing the mobile applications have appeared recently. One of them is React Native [29], which is a framework for building native mobile applications using Javascript. The main advantage of this framework is its capability of creating cross-platform applications, for both of the mentioned platforms. Although the technology such as Javascript is known mainly from the web development, the resulting application does not run in a web view - a mobile browser. React Native uses a middleware called „bridge“, which communicates with the native components, written in Java or Objective-C. Therefore, the resulting application is indistinguishable from an application built using the traditional ways. The usage of this framework can also bring some disadvantages. The support for Android and iOS has been released only in 2015, and therefore, the documentation, as well as the community are still developing. Some of the native APIs are not accessible yet, however in case of need, React Native can still be combined with the native code.
The working principles of React Native are very similar to React [28], a framework used for web development, with the exception of not using HTML, but the corresponding alternative components. The main building block of React is a `React.Component`, that is used to split the user interface into independent and reusable pieces. Every `React.Component` has its own lifecycle, and the corresponding lifecycle methods, called at particular times. A few examples of lifecycle methods are `constructor()`, `componentDidMount()`, but most importantly a method called `render()`, which is required and returns an output of every component. This method is called every time a `React.Component` updates, which can be caused by changes in its properties - props, the component's state or the update of parent component.
With the popular trend of creating single-page JavaScript applications, the requirements for managing internal state has grown up. The state usually contains server responses, as well as locally created data. Additionally, it is often used for managing the state of user interface, for example active routes, selected tabs, spinners, pagination controls, and so on.

Redux [31] is a JavaScript library, which is often used for this purpose in combination with React or possibly, React Native. It is build around three core principles which are:

- **Single source of truth** - The state of the whole application is stored in an object tree within a single store. The single store, therefore, can contain both client and server data.
- **State is read-only** - The only way to change the state is to emit an action, an object describing what happened. All the actions are centralized and happen one by one in a strict order, and therefore, there are no subtle race conditions.
- **Changes are made with pure functions** - To specify how the state tree is transformed by actions, it is needed to write the reducers. Reducers are pure functions that take the previous state and an action, and return the next state.

Redux works well with React mainly because of the fact, that React implements its components as a function of state, updated whenever the state has changed. Since Redux emits the state changes as a response to its actions, the components are updated accordingly. React Native and Redux have been used to implement the client side of the application.

5.2.2 Client Components

The client application has been divided into several components, which are described in this section. Every component corresponds to the actual implemented `React.Component` in the application. The components are be described according to their relations in the component tree in Figure 5.2, starting from the parent.

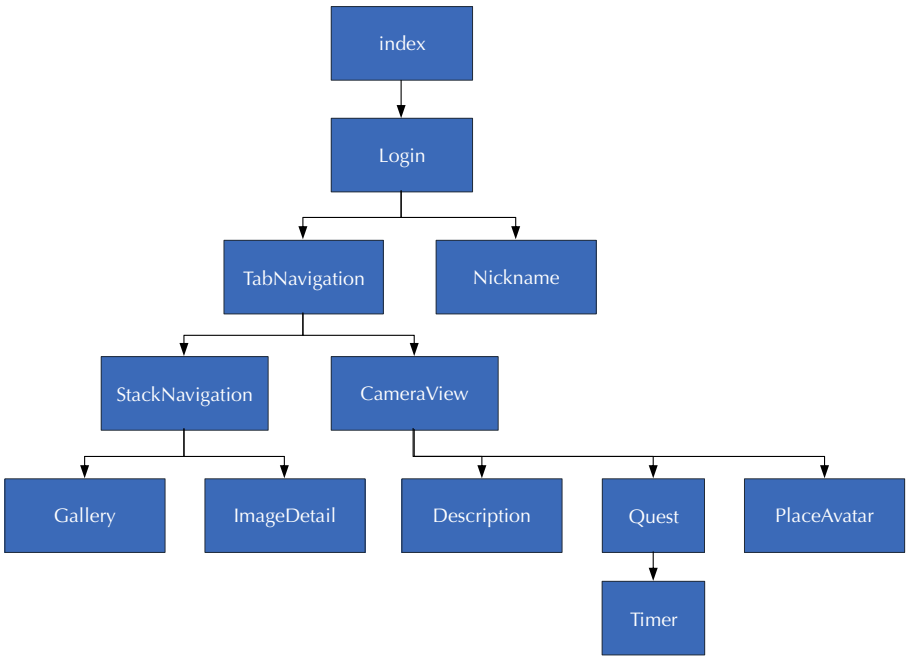


Figure 5.2: A component tree in React Native

- **`index`** - a top-level parent component, which is called when the application is started. This component creates the Redux store with its initial state, and provides it to all the children components, using `Provider`. The children components can, therefore, access the store using `connect()` function.

- **`Login`** - the first visible component. It is responsible for user authentication, described in Section 5.2.3. If the user is not authenticated, a log in screen is shown. Otherwise, the children components are rendered, while showing a loading screen. The component also sets the initial database listeners, and changes the Redux state accordingly. Additionally, because of its position above all the components, it checks the network connection, and shows an offline screen anytime it is not available.

- **`TabNavigation`** - this component creates a tab navigation, which has been originally proposed in Chapter 2. The tab navigation contains two tabs labeled Gallery and Quest. The top components used for the corresponding tabs are `StackNavigation` and `CameraView`, both containing further component subtree.

- **`Nickname`** - a component for entering user nickname. It is used whenever a user logs in, and the nickname for his profile is not set in the database. After entering the nickname, the user will never see this component again.

- **`StackNavigation`** - a stack navigation inside Gallery tab, navigating between `Gallery` and `ImageDetail`.

- **`CameraView`** - a component responsible for accessing a camera and taking a picture for the given quest. The component splits the screen into two parts, showing the camera view or the taken picture on the upper part of the screen, and `Quest` component at the bottom. The camera view, as well as the taken picture, have a square shape. The reasoning behind it is discussed in Section 5.3.3.
  Before accepting and after completing the quest, the component displays `Description` and `PlaceAvatar` components inside the original camera view.

- **`Gallery`** - a component showing the images of all completed quests, fetched from the database. The images are shown inside of `FlatList`, which enables scrolling as well as dynamic scroll loading. Whenever an image is clicked on, it navigates to `ImageDetail` component.

- **`ImageDetail`** - the detail of the completed quest. The component shows the full image, the items of the given quest, and gives the user a possibility to share the image on other social media.

- **`Description`** - a simple component showing a text description for the new quest.

- **`Quest`** - a component which accompanies the user throughout the process of completing quest. It changes its content according to the current state of the quest, starting from its acceptance, when the request is send to the server and a new generated quest is received, and sent to the database. After that, the component shows the list of items for the current quest, the button for taking a picture, and `Timer` component. When a picture is taken, it is first sent to the database storage, described in Section 5.3.4. The image URL is returned and a request for evaluation is sent to the server. Meanwhile, a circle loading animation is shown. On successful completion of

the quest, the component contains the arrows for changing the avatar type, and a confirmation button. If the completion was not successful, a repeat button is shown.

- `PlaceAvatar` - a component used for placing the animated avatar inside the picture. The position of the avatar can be moved around the picture using touch. When the placing is confirmed, a snapshot of the view containing the image and the avatar is taken, and sent to the database.

- `Timer` - a component showing the remaining time for the quest. When the time runs out, the component deletes the actual quest from the database, which changes the state of the application and rerenders the components to the original state of accepting the quest.

5.2.3 Authentication

This section describes the implementation of authentication, from the client's point of view. There are several ways of authenticating user - a basic email and password authentication, as well as using federated identity providers, such as Facebook, Google or Twitter. In an ideal scenario, the user should have an option to choose the way of authenticating. However, not all the options could be implemented in the scope of this project, and therefore, the usage of Facebook Login has been chosen as the main way.
The implementation of Facebook Login is done through The Facebook SDK, provided for different platforms. In case of React Native, a wrapper called `react-native-fbsdk` exists. The functionality of Facebook Login is based on the access tokens. A process of getting an access token is shown in Figure 5.3. The application first requests access via SDK, the user authenticates and approves needed permissions, and a temporary access token is provided to the application. There are several types of access tokens, such as User Access Token, App Access Token or Page Access Token, used for different API calls. In this application, Facebook Login was used only as a way of user authentication, without any integration with actual Facebook, such as posting or reading user data. Therefore, only the most common User Access Token has been used. The default persmissions are `email` and `public_profile`, which can be used without submitting the application for a further review. The received access token has been integrated with Firebase Authentication, described in Section 5.3.4

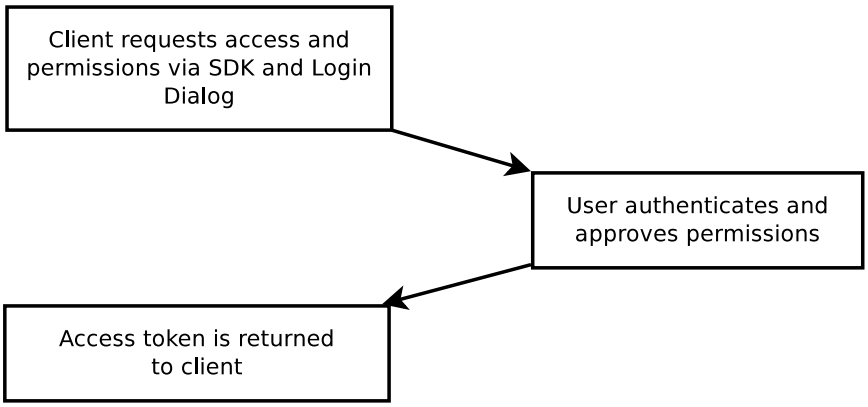


Figure 5.3: Process of getting an access token via Facebook Login. Based on

5.2.4 User Interface

The development of the user interface in React Native, is similar to styling in web development. Every core component in React Native accepts a property named style, which is a Javascript object, containing the given styles. The style names and values are similar to Cascading Style Sheets (CSS), just written using camel casing. The main layout of the user interface has been done using Flexbox and Dimensions, taking into consideration the different device resolutions. An example of a style using Flexbox, that has been used in `Login` component, can be seen in Listing 5.1.

```
introScreen: {
       flex: 1,
       flexDirection: 'column',
       alignItems: 'center',
       justifyContent: 'center',
       backgroundColor: 'white',
}
```

Listing 5.1: A style example used in `Login` component.

The user interface has been built according to Section 2.2, and the wireframes proposed in Figure 2.2. Figure 5.4 shows the user interface of `Login`, `Gallery` and `CameraView` components, with their corresponding subcomponents. The navigation between `Gallery` and `CameraView` components is displayed at the bottom, corresponding to `TabNavigation` component. The navigation action can be triggered both by swiping left or right, as well as clicking on the tab buttons.

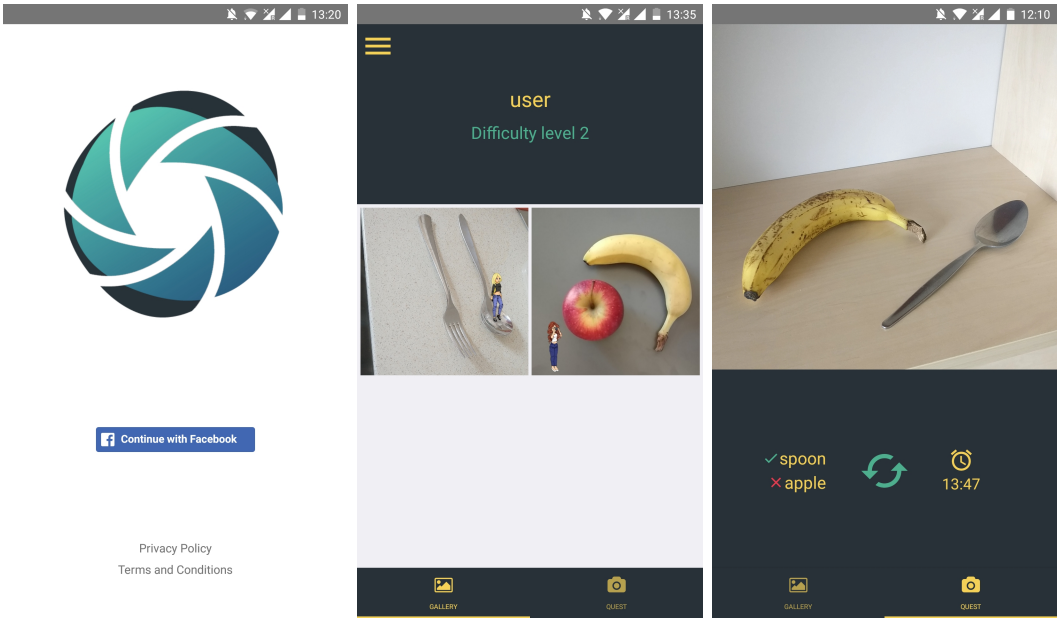


Figure 5.4: The user interface of the application, matching the wireframes originally proposed in Figure 2.2.

The other actions such as showing `ImageDetail` component, or the user interaction when completing the quest, also work according to the initial description. These components have a very similar user interface, and some of them can be seen in Appendix B and C. The colors of the user interface have been set in an experimental way, and can easily changed if required.
Additional graphics was required, in order to match the required user experience. This included the animated avatars, and the logo of the application. The graphics has been created by people mentioned in Acknowledgements section, directly for the purpose of the application. The logo has been used as a miniature icon for the application, as well as in `Login` component, shown on the first screen in Figure 5.4. The animated avatars have been used in `PlaceAvatar` component, and are shown in Figure 5.5.

Figure 5.5: The animated avatars in the application.

5.3 Server Side

According to the proposed technical design, the server is responsible for the quest assignment, image recognition and storage. The server is built using two separate parts, responsible for these roles. These are the logical part, presented as the server, and the database.

5.3.1 Communication

The communication between the server and client has been done through HTTP requests, using a simple API and JSON [18] data format. The API has been built using Flask [11], having two basic routes, shown in Table 5.1. These correspond to the quest assignment and the inference of the image recognition model.
The role of quest assignment, is accessible at `/quest?level=[number]` route using `GET` method, where the `[number]` is a difficulty level, for which a quest is needed to be assigned. The returned JSON object contains an array of item IDs, item names and the ending time of the quest. Its functionality is further described in Section 5.3.2.
The second role, the inference of the image recognition model, is accessible at `/model` route, using `POST` request and `application/json` content type. The required parameters for the

request are image and an array of item IDs. The image parameter contains a URL of the image, which is used as an input to the model. The second parameter, the array of item IDs, corresponds to the array returned by the quest assignment, containing the IDs of items to be recognized in the image. This role is closely described in Section 5.3.3.
The implemented architecture does not contain any direct communication between server and database, except of downloading the image from the database storage, using publicly available URL. The rest of the communication is always done through the client, which makes a request to the API, and saves the needed parts of the response to the database.

URI	Method	Input	Output
`/quest?level=[number]`	`GET`	-	`itemIds: [array], itemNames: [array], endTime: [string]`
`/model`	`POST`	`image: [string], itemIds: [array]`	`results: [array], completed: [boolean], cheating: [boolean]`

Table 5.1: The routes of the server API, with the corresponding parameters.

5.3.2 Quest Assignment

A total of 26 object classes (29 object classes trained, 3 object classes used to detect cheating) have been divided into 7 difficulty level categories. Additionally, a time frame has been experimentally set for every difficulty level. Both object classes and time frames for all the levels are shown in Table 5.2.
When a request is received, two non-repeating items - object classes are generated for the level, specified by the request parameter. If the level is higher than 7, meaning that the user has already passed all the basic difficulties, the items are generated from the whole set of object classes, which is meant to introduce new combinations and still provide a content for the application. The ending time is assigned in UTC time format, adding the time frame specific for the given level, to the current UTC time. The response is sent back to the client, according to the previously mentioned API, and the client saves the quest in the database.

Difficulty level	Object classes	Time frame
1	fork, spoon, banana, apple	15 minutes
2	book, toothbrush, cup, scissors	30 minutes
3	backpack, clock, cake, donut, chair	45 minutes
4	broccoli, tennis racket, tie	60 minutes
5	pizza, frisbee, orange	75 minutes
6	bicycle, skateboard, teddy bear	90 minutes
7	cat, dog, fire hydrant	105 minutes

Table 5.2: A table showing the time frames and sets of object classes for every difficulty, from which a quest is generated.

According to the logic of the application, a request to generate a new quest is sent when the user has completed a previous quest, as well as when he failed - the time has run out. In this case, there is a chance of generating the same quest as before, which could be prevented. The functionality would require sending an information about the previous list of quest items to the server. Unfortunately, this functionality has not been implemented yet.

5.3.3 Image Recognition

Although in Section 6.1, the object detection models show better performance, the multi-label classification model has been chosen for the purpose of this application. The object detection models have been disregarded, because of their high computational complexity. On GPU, the speed of Faster R-CNN reaches 5 fps, and the speed of Single Shot MultiBox Detector reaches 59 fps. The model of Single Shot MultiBox Detector could potentially be considered, however, the operation costs of using a GPU on the server are high, and running it on CPU would slow down the inference even further, which is unacceptable from the user experience point of view. Another advantage of using the multi-label classifier is that its input training images do not require having bounding boxes. This makes the model more flexible and scalable, in case the user images are used for its improvement.
A recieved request for model inference contains the URL of the input image, which is first download from the database storage. The downloaded image is preprocessed and rescaled to the target size of 299 per 299 pixels, which is the same size as used in training. The original images already have a square shape, which has been implemented by the client side. Therefore, rescalling the image does not skew its proportions, which could potentially decrease the model accuracy.
When the input image is in correct format, it gets evaluated by the model, and a list of scores for all trained object classes is returned. The IDs of object classes, which should be contained in the image, have been recieved in the request as `itemIds`. If score for the object class, given received ID is higher than the threshold, its result in the corresponding `results` array is set to true. The threshold has been set to 0.25, according to Section 6.1. The parameter `completed` is set to true, in case all the recieved object classes have been successfully evaluated. Additionally, a check for cheating items is performed. These are laptop, cell phone and television object classes. If any of the quest items is successfuly evaluated and a cheating item has been found, the parameter `cheating` is set to true.

5.3.4 Firebase

Firebase [10] is a platform, that is often used in mobile application development. It offers several tools, not only for building the application, but also for its subsequent analysis. Another advantage of this platform is its scalability, according to current traffic load and needs of the application. The tools from Firebase, that have been used in this project are Realtime Database, Cloud Storage, and Authentication.
Firebase Realtime Database is a cloud-hosted database, where data is stored as JSON and synchronized in realtime to every connected client. It uses data synchronization every time data changes, where the connected devices might use listeners and receive updates within milliseconds. This works especially well with Redux and React Native. On data change, the listener can dispatch the appropriate action to change Redux state, which from the functionality of React, rerenders the needed parts of the application.
In this project, Firebase Realtime Database has been used to store the user data. An Entity

Relationship Diagram in Figure 5.6 shows the relationships between the entities stored in the database. Every user is represented by the entity `User`, which has a unique id, and the attributes nickname and level. The actual ongoing quest is represented by the entity `Actual`. This entity is in `0..1 to 1` relationship with the entity `User`, meaning that the user does not need to have any actual ongoing quest, for example if the time has already run out, or the quest has not been generated yet. Additionally, every actual quest is assigned to exactly one user. The `Actual` entity contains the end time of the quest, and the arrays of item ids and item names. The quests that have been already completed are represented by the entity `Quest`. Every user could have completed zero or more quests, and every quest was completed by exactly one user. This is represented by its `1 to 0..N` relationship with the entity `User`. Every completed quest has a unique id, the arrays of item ids and item names, and the image URL, linking its image path in the storage.

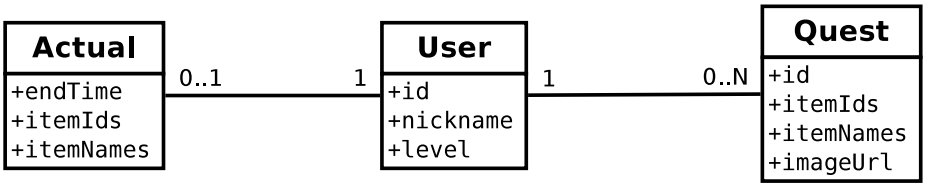


Figure 5.6: An Entity Relationship Diagram illustrating the logical structure of the database.

Cloud Storage for Firebase is made with the intention to store and serve user-generated content, such as images or videos. In case of this project, it has been used to store and serve the pictures taken by the users, to complete the quests. The storage has been divided into two main folders, `dataset` and `users`. The `dataset` folder is used to store every picture, that has been taken and sent for evaluation, with the intention to use the pictures to improve the model in the future. The pictures are sorted into subfolders, according to the quest items they have been taken for, which is meant to simplify the further picture labeling. The subfolders might be, for example `toothbrush_scissors` or `backpack_chair`. The `users` folder contains a subfolder for every user, named according to their ID. For now, the subfolder contains the completed quests of every user. These are the pictures already containing the animated avatar, and linked by the image URL in `quest` entity of the database.

```
{
    "rules": {
        "users": {
            "$uid": {
                ".read": "$uid === auth.uid",
                ".write": "$uid === auth.uid"
            }
        }
    }
}
```

Listing 5.2: Rules in Firebase Realtime Database using Firebase Authentication

The last service used from Firebase is Firebase Authentication. This service is responsible for authenticating user. It is integrated with other Firebase services, and can be used to secure the saved user data in Realtime Database or Cloud Storage. The service supports a basic email and password authentication, as well as integration with federated identity providers, such as Facebook, Google or Twitter. As mentioned earlier, Facebook identity provider has been used for this purpose. The authentication assigns the `User UID` to every signed user. The `User UID` can used as a unique ID in the database or storage, and the simple rules for securing data can be written. The example rules that have been used to secure the database, are shown in Listing 5.2.

5.3.5 Deployment

Since the game logic of all clients is centralized and the number of clients can grow over time, the server has to be able to handle multiple incoming requests at once. The possible problems could occur mainly with the image recognition part, which has high computational complexity. This part has been solved by creating a distributed architecture shown in Figure 5.7, using Celery [3].
Celery is an asynchronous task queue based on distributed message passing. It uses execution units called tasks, which are executed on a single or more workers. The tasks can be executed asynchronously in the background, or synchronously by waiting for them to finish. The communication between clients and workers is done through messages, using a broker.

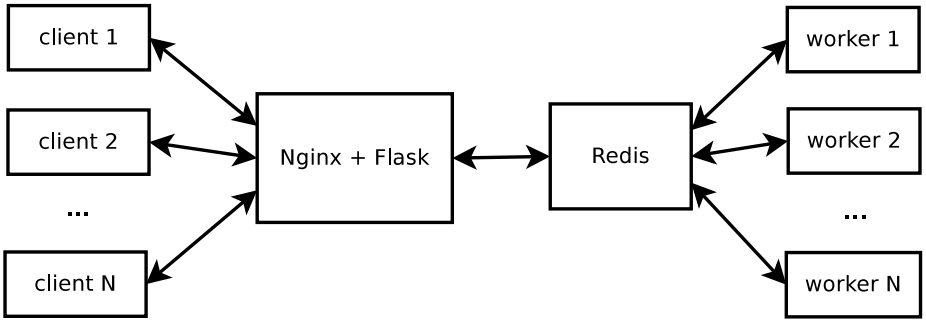


Figure 5.7: The overall distributed architecture.

As mentioned, the task that has been deployed using Celery is image recognition - inference of the model. On startup, a created worker loads the model and waits for messages from the broker, which in case of this project is Redis server [30]. When a request from the client is recieved in the API described in 5.3.1, a task message is sent to the broker, and the request synchronously waits for the task to be executed. The message contains data needed for the execution, according to Section 5.3.3. The transferred data needs to be serialized, which is the reason why image URL has been used as the parameter of the API, instead of the image itself. This way, the data can be kept in JSON format, making the communication more lightweight. After the worker has finished executing the task, the result is sent to the waiting request through the broker, again in JSON format. The waiting request sends the response back to the client.
The configuration of Celery has been set according to the description. The parameters

such as `CELERY_BROKER_URL` and `CELERY_RESULT_BACKEND` have been both set to the address of Redis server, running on port 6379. The parameters `CELERY_ACCEPT_CONTENT`, `CELERY_RESULT_SERIALIZER`, and `CELERY_TASK_SERIALIZER` have all been set to JSON. The running worker uses single-threaded execution pool, which is specified by the parameter `-pool=solo`. The worker has been deployed in a docker container [9], which has been built using Dockerfile shown in Listing 5.3. An advantage of this approach is that by creating a docker image from the container, the scaling and deployment of any number of workers gets easier.
The mentioned API, which has been created as a Flask application, is served using Nginx as a reverse proxy. It has also been deployed in a docker container, using `uwsgi-nginx-flask-docker` [36] image. The containers have been deployed on Amazon Web Services (AWS) [1] platform, using EC2 instances.

```
FROM python:2.7
RUN pip install -U pip
RUN pip install flask celery numpy tensorflow keras redis
COPY ./app /app
WORKDIR /app
ENTRYPOINT celery -A main.celery worker --loglevel=INFO --pool=solo
```

Listing 5.3: A Dockerfile for Celery worker

Chapter 6

Evaluation

This chapter evaluates the trained image recognition models, as well as the application itself by user testing.

6.1 Image Recognition Models

Besides the metrics used during training, the image recognition models needed to be further evaluated. The reason for this was to determine, which approach achieves better performance, as well as to set the threshold for recognizing objects in the quest pictures.
Considering the user experience and gameplay of the application, there are two scenarios that could go wrong, when using image recognition. The first is that the application recognizes object that is not in the picture, meaning that the user is able to fool the application. The second scenario is that the object is in the picture, and does not get recognized. These scenarios match the cases of false positives and false negatives, which along with true positives and true negatives are the basis of the evaluation metrics used in this Section.
The evaluation of all metrics and models has been done on validation part of COCO: Common Objects in Context dataset, considering only the 29 trained classes. Futher evaluation on user images directly from the application could provide additional value, since these images can have some specific characteristics. Unfortunately, it could not be achieved, for the reason of having insufficient amount of images with a low variety of object classes, described in Section 6.2.

6.1.1 Thresholding

Most of the classifiers output scores for object classes, without setting a threshold for determining, whether the object class is, or is not contained in the instance. The threshold can, therefore, be set according to the needs of the given system. In general, by increasing the threshold, the number of false positives decreases, while the number of false negatives increases. In case of the application developed in this project, the threshold has a direct influence on user experience.
In case of object detectors, the output along with the scores contains the detected bounding boxes, which can also be thresholded. As mentioned in Section 5.1.2, during training Single Shot MultiBox Detector and Faster R-CNN reached Mean Average Precision (mAP) of 0.45@0.5IOU and 0.54@0.5IOU. Mean Average Precision is the mean over average precision of multiple or all classes, calculated in the next section 6.1.2. The IOU stands for Intersection Over Union [17], which is a ratio between the intersection and the union of

the predicted boxes and the ground truth boxes, illustrated in Figure 6.1. The Intersection Over Union is used as the mentioned threshold, for the accuracy of the detected object location. As mentioned in Section 4.5, the information about object location is not needed for the purpose of the application, since the object detectors have been used as classifiers. The thresholding using Intersection Over Union could, therefore, be disregarded. This decision increased mAP of Single Shot MultiBox Detector from 0.45@0.5IOU to 0.7771, and mAP of Faster R-CNN from 0.54@0.5IOU to 0.7835. The rest of the metrics in this Section, have been all evaluated without Intersection Over Union thresholding for object detectors.

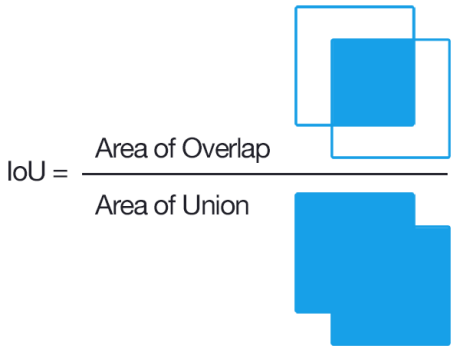


Figure 6.1: An illustration of Intersection Over Union, taken from [17].

6.1.2 Precision-Recall

Precision-Recall [26] is one of the metrics, that has been used to evaluate the models. It is a measure of relevance that comes from Information Retrieval, where precision is defined as the fraction of relevant instances among the retrieved instances, and recall is defined as the fraction of relevant instances that have been retrieved over the total amount of relevant instances. There is a trade-off between precision and recall, meaning that if fewer instances are retrieved, the precision increases while recall decreases, a vice versa, if for example all instances have been retrieved, recall is equal to 1, while precision is very low.
Relating to the concept of true/false positives/negatives, a high precision can be interpreted as representing a low false positive rate, and high recall as representing a low false negative rate. Precision can be calculated as the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p), shown in Equation 6.1. Recall is calculated as the number of true positives over the number of true positives plus the number of false negatives (F_n), shown in Equation 6.2.

$$Precision = \frac{T_p}{T_p + F_p} \tag{6.1}$$

$$Recall = \frac{T_p}{T_p + F_n} \tag{6.2}$$

The metric is usually displayed as a Precision-Recall curve, which shows the mentioned tradeoff between precision and recall for different thresholds. The area under the curve can

be interpreted as Average Precision (AP) of the given object class. The Precision-Recall curve has been averaged over all object classes, and created for every model (shown in Figure 6.2). This way, the area under the curve relates to Mean Average Precision (mAP), which is 0.6802 for Multi-label Classifier, 0.7771 for Single Shot MultiBox Detector and 0.7835 for Faster R-CNN. Comparing the object detectors, Faster R-CNN achieves only a slightly better performance than Single Shot MultiBox Detector. However, their performance is considerably better than the performance of Multi-label classifier. The Average Precision for every trained object class can be found in Appendix A.

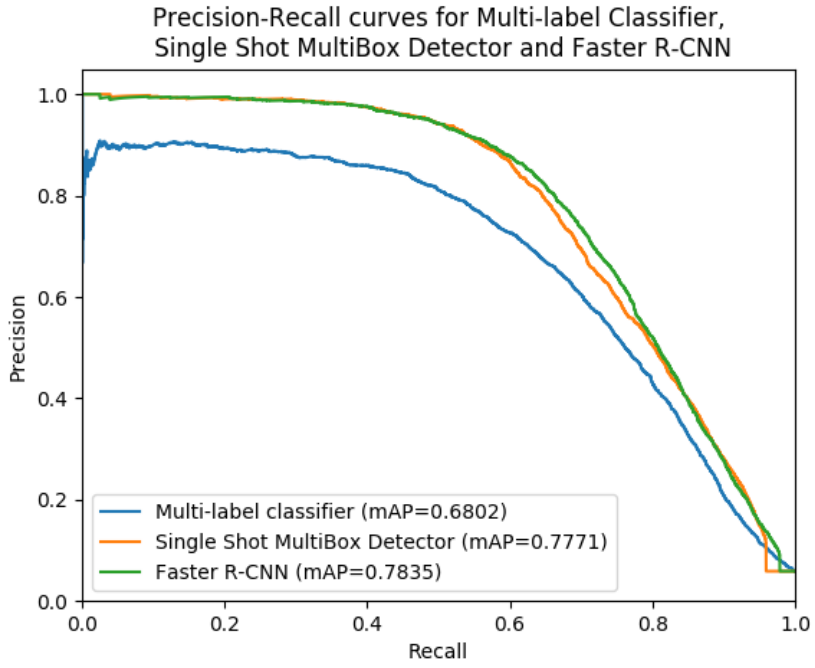


Figure 6.2: A Precision-Recall curve averaged over all object classes, evaluated for every trained model.

6.1.3 Custom Metric

In order to have a different look on the trained models, as well as to get more useful information, another custom metric has been created created. Since the classification is done in a multi-label setting, the number of false positives and false negatives can be higher than one, and may vary for every instance. This metric shows the percentage of cases, given the number of false positives and false negatives. The resulting percentage of one is independent from the other, meaning that the percentage of false positives does not take into consideration false negatives, and vice versa. The sum of percentages over all numbers makes up 100%, for both false positives and false negatives. The threshold at which the metric has been evaluated has been set to 0.25, which for the multi-label classifier used in Section 5.3.3, gives an acceptable and balanced number of false positives and false negatives, considering the user experience. The metric has been evaluated for every model.
The evaluation of multi-label classifier shows, that in 54% of instances there is not any false positives and in 58% there is not any false negative. The cases with one false positive or one false negative both make up 30%. The rest of instances mostly have 2 false positives or

false negatives. A graph containing the results of multi-label classifier is shown in Figure 6.3.

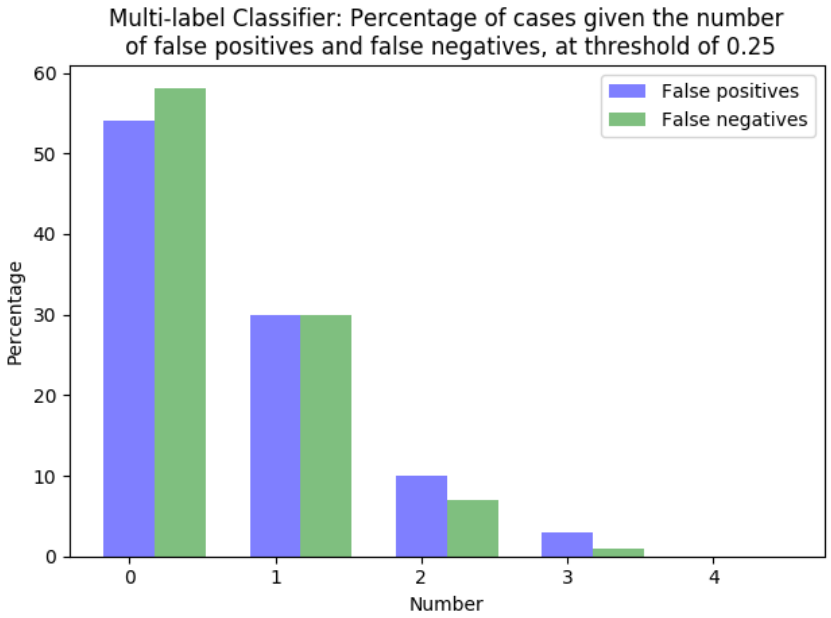


Figure 6.3: Evaluation of the custom metric for Multi-label Classifier.

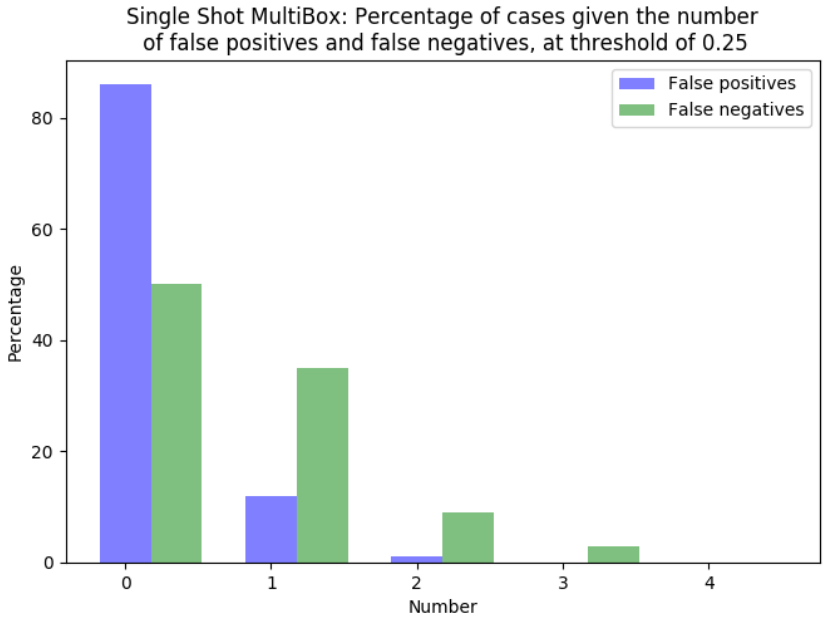


Figure 6.4: Evaluation of the custom metric for Single Shot MultiBox Detector.

The evaluation of Single Shot MultiBox Detector is shown in Figure 6.4, and shows a different results. No false positive can be found in 86%, while no false negative only in 50% of cases. The reason for this difference is caused by having a low threshold. A single false

positive can be found in 12% of cases and single false negative in 36% of cases. The rest of instances have mostly two false negatives, while the percentage of having more than one false positive can be disregarded.
At last, Faster R-CNN has been evaluated using this metric. Unlike for Single Shot Multi-Box Detector, the threshold for this method seems to be more balanced. The number of cases with zero false positives is equal to 70%, and the number of false negatives is equal to 60%. The percentages of having one false positive or false negative are 22% and 30%. The rest of instances mostly have two false positives or false negatives.
According to this metric, it is possible to see different results for every trained model. Unlike in previous Section 6.1.2, the performance of object detectors differs, however, it still prevails over multi-label classifier.

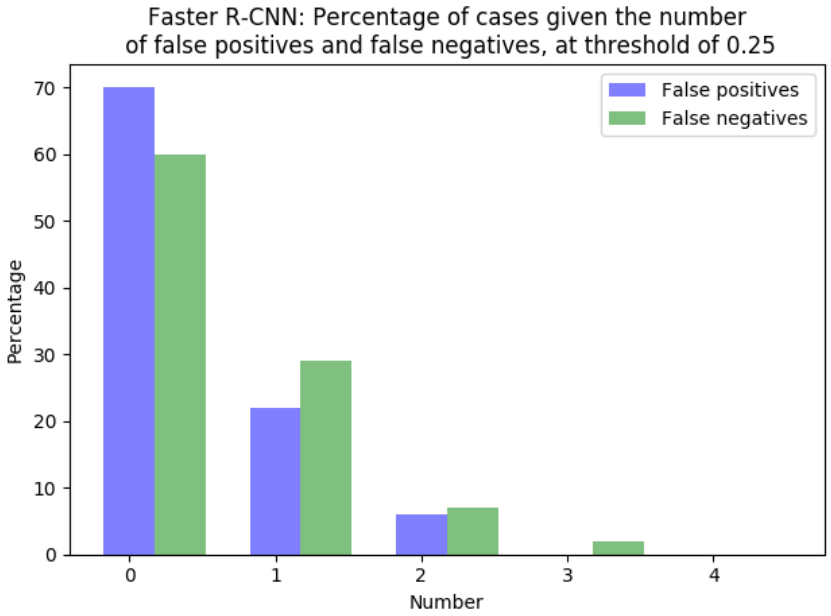


Figure 6.5: Evaluation of the custom metric for Faster R-CNN.

6.2 User Testing

The user testing has been done in two phases, before and after release. Both phases and their provided values are described in this section.

6.2.1 Beta Testing

Before release, the application has been distributed to 10 users, in order to test the concept and get a feedback about the initial user experience. The information about their progress in the game, as well as the taken pictures were gathered in the database. Additionally, the users were personally asked about their overall experience, possible issues and ideas.
The main issue that has occurred, was having quests with the list of 3 items, as originally designed in Chapter 2. This caused several complications. It was difficult to put all the items into one picture, since they may vary in size, and the camera frame in the application

had a square shape, explained in Section 5.3.3. Additionally, even when the users managed to fit all the items into the picture, the recognition part did not work as planned, and usually only correctly classified 2 out of 3 items. This was mostly caused by the threshold for approving items in the image. The confidence of the model for the items was often spread out, and the third item was usually below the threshold. Lowering the threshold, which has already been set to 0.25, would cause having more false positives, which was undesirable. Therefore, the decision to have only quests of 2 items was made. This decision also brought a possibility of having more quests and difficulty levels along the way, since only 29 classes have been trained.
The user interface has been tested as well, mainly measuring the time from the first log in into the application, to the time a user gets to the Quest view and generates his first quest. This was on average 13 seconds. Considering the fact that the users have not seen the application before, together with the time it takes to read the quest instructions, this result seemed to be satisfactory. Therefore, no further changes to the user interface have been made.
The authentication using Facebook Login, described in Section 5.2.3, had some negative comments from the users, mainly regarding the user privacy. For this reason, as well as other legal reasons, Privacy Policy [1] has been written. Providing additional ways of authentication, could also help with this issue. Other possibility would be to remove authentication completely, which would disregard its added value discussed in Section 3.2.1. In the end, no changes to authentication have been made before release, however, its influence on user behaviour needs to be further observed.
Overall, the tested users were interested in the idea of the application, and on average, completed quests from 3 difficulty levels. An example of a picture from beta testing can be seen in Figure 6.6. The items in the next levels were more difficult to obtain, therefore, some users either stopped playing or tried to cheat. For this reason, a functionality to detect some cases of cheating, described in Section 5.3.3, has been created, and other future improvements are described in Section 7.2.

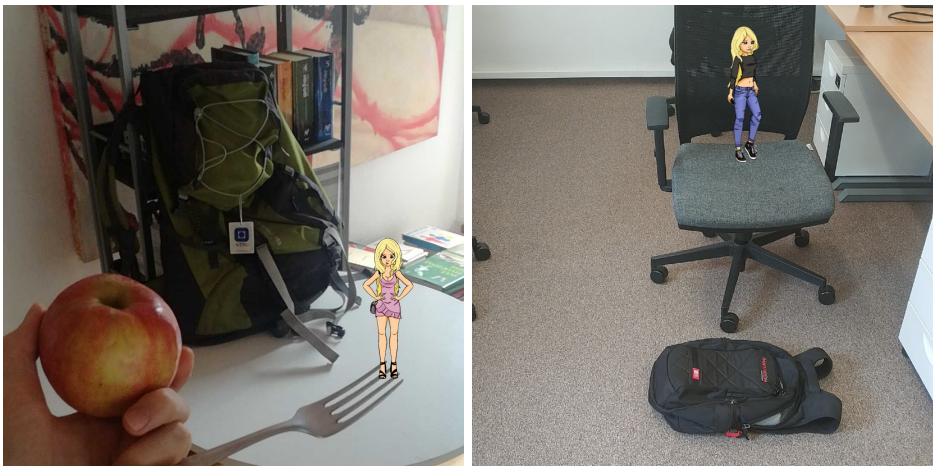


Figure 6.6: The quest pictures taken by the actual users. The picture on the left has been taken during beta testing, for the quest fork, backpack, apple (still 3 items). The picture on the right has been taken during production, for the quest backpack, chair.

[1] available at http://stageupgame.com/privacy_policy.html

6.2.2 Production

After the end of Beta testing phase, the main implementation issues have been fixed, the server has been deployed, and the application was ready for the first release. Although React Native, described in Section 5.2.1, is able to build cross-platform applications and the server side is platform independent, the application has been tested mainly for Android devices. Therefore, the first version of the application has been published only for this platform, on Google Play. The application has been published under the name „StageUp“, which refers to staging up the items for the picture. A listing of the application on Google Play can be seen in Appendix C.
Currently, the application has 21 authenticated users, with the highest difficulty level being level 4. So far, the application has been rated by 7 users, all giving it a 5 star rating, shown in Figure 6.8. Additionally, a few comments have been added together with the rating [2], most of them being positive or regarding the issue with book object class discussed below. There have been 58 pictures taken and evaluated so far, which are currently stored in the database. From these pictures, only 16 have been successfully evaluated to contain all the items, and saved in the database as completed, together with the placed animated avatar (one of them is shown in Figure 6.6). Most of the pictures have been taken for the first few difficulty levels, therefore, the variety of object classes, as well as the total number of pictures is still low for the model evaluation.

Figure 6.7: On the left, an instance containing book object class from COCO dataset. On the right, a user picture for the quest book, scissors, which has not been successfully evaluated.

The low ratio of successfully evaluated pictures compared to all the pictures, can mean several things. The users are either taking pictures without the correct objects, or the application is not able to recognize all the objects in the picture. From looking at the pictures in the database, it can be seen that both options are represented. There are pictures that do not contain any required objects, which can be explained as users just trying the functionality of the application. However, there are also pictures that should have been evaluated as completed, which was not the case. Particularly, there was a problem in recognizing book object class, which is an item at difficulty level 2. According to Appendix

[2] application is available at https://play.google.com/store/apps/details?id=com.stageup

A, the average precision of this class is 0.57. This is higher than the average precision of object classes such as spoon - 0.39 or apple - 0.51, which are part of the first quest, and have been successfully evaluated most of the time. The reasoning behind this is difficult to interpret. One of the possible interpretation could be based on looking at the images of book object classes in COCO: Common Objects in Context dataset, on which the model has been trained. Many instances of this object class in the dataset contain several books, usually placed in the shelf or on top of each other, compared to a single book being placed in the front, which is usually the case in quest images. An example is shown in Figure 6.7. Another issue that has been discovered during production, is regarding server security, which has not been taken into consideration in this project. However, it could cause problems in the production environment, since the API has been made publicly available. Looking at the logs from the server, many requests have been made to the server, which do not come from the clients using the application. So far, all of them have been rejected, for the reason of incorrect input. The solution for this issue is proposed in Section 7.2.

Figure 6.8: The user rating on Google Play.

Chapter 7

Conclusion

7.1 Summary

This thesis has successfully created a unique photo-challenge mobile application, according to the initially proposed idea. Its creation and development process has been fully described, while addressing the reasons for most of the decisions that have been made. The initial idea proposal was followed by the product and technical design, implementation, and evaluation. The thesis has also focused on the image recognition, which is a core functional part of the application. The background of image recognition has been explored, finding the most suitable options for the application. These were object detection and multi-label classification, which have been both implemented, evaluated and suited according to the needs of the application. Although object detection seems to achieve a better performance, multi-label classification has ended up being used, for the reasons of operation cost.
The resulting application can bring value from both, user and technical point of view. The user experience has been designed in consideration with the current state of the art, with the goal of making the application interesting. The actual usage of the application may create a new multi-label image dataset, which can be used to further improve the used image recognition models, and with the approval of the users, can be published for the community.
The application is currently available on Google Play under the name StageUp, and the work has been presented on the conference Excel@FIT 2018 [1], where it received a committee reward for creation of a quality platform for data gathering.

7.2 Future Improvements

As with any project, there are many features and improvements, which could be further introduced, some of them being already addressed in Section 6.2. The published application is currently in its first version, and although it is fully functional, it can still be considered as a minimum viable product.
The main possible improvements could be done on the client side of the application, which has been considered secondary in this project. One of them is to introduce user notifications, triggered when the time limit is running out and a new quest can be generated. This would engage users, which tend to quit the application after several levels, and not return again. The other possibilities on the client side would be to add more content, by adding more

[1] http://excel.fit.vutbr.cz

animated avatars, and the actions that users can take. These could include options to customize the avatars, get rewards after completing the quests, or play mini games with the avatars. Adding these features would break the user from the stereotype of just taking pictures, or the opposite, motivate the user to take even more pictures and receive rewards. At last, according to the current structure of the server and database, the options to create a community, mentioned in Section 2.2, could be implemented mostly on the client side.
More content could also be introduced on the server side, in a sense of training the models to recognize more object classes, used for generating quests. This would require to find new images, since all the suitable object classes from COCO dataset have been already used. The other datasets have been disregarded mainly for not containing instances with multiple object classes, needed to train a multi-label classifier. This problem could be solved, if the object detectors are used for the functionality of image recognition, which would according to Section 6.1, also result in higher precision. However, the models still need to be evaluated on the user images.
The security issue on the server side, that has been discovered in Section 6.2.2, could be solved by introducing HTTP over TLS [13], as well as by confirming the requests by sending and checking the user access token. This will probably be one of the first steps, that will be taken.
To make the application more popular and recieve more feedback, additional marketing is required, since the application is currently quite difficult to discover. If the application gets popular, there is a possibility of monetizing and building it further on.

Bibliography

[1] Amazon Web Services (AWS). [Online; accessed 20.05.2018].
Retrieved from: https://aws.amazon.com

[2] App Store. [Online; accessed 20.05.2018].
Retrieved from: https://www.apple.com/lae/ios/app-store/

[3] Celery: Distributed Task Queue. [Online; accessed 20.05.2018].
Retrieved from: http://www.celeryproject.org

[4] Clarifai. [Online; accessed 20.05.2018].
Retrieved from: https://www.clarifai.com/technology

[5] Client-server model. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Client%E2%80%93server_model

[6] Computer Vision. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Computer_vision#Recognition

[7] Convolution. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Convolution

[8] Convolutional Neural Networks (CNNs / ConvNets). [Online; accessed 20.05.2018].
Retrieved from: http://cs231n.github.io/convolutional-networks/

[9] Docker. [Online; accessed 20.05.2018].
Retrieved from: https://www.docker.com

[10] Firebase. Google Developers. [Online; accessed 20.05.2018].
Retrieved from: https://firebase.google.com

[11] Flask. [Online; accessed 20.05.2018].
Retrieved from: http://flask.pocoo.org

[12] Google Play. [Online; accessed 20.05.2018].
Retrieved from: https://play.google.com

[13] HTTPS. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/HTTPS

[14] Importing Data. TensorFlow. [Online; accessed 20.05.2018].
Retrieved from: https://www.tensorflow.org/programmers_guide/datasets

[15] Instagram. Google Play. [Online; accessed 20.05.2018].
Retrieved from:
https://play.google.com/store/apps/details?id=com.instagram.android&hl=cs

[16] iOS. Apple Developer. [Online; accessed 20.05.2018].
Retrieved from: https://developer.apple.com/ios/

[17] Jaccard Index - Intersection over Union. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Jaccard_index

[18] JavaScript Object Notation. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/JSON

[19] lifeshot – Photo Challenge App. Google Play. [Online; accessed 20.05.2018].
Retrieved from: https://play.google.com/store/apps/details?id=com.keepinmind.lifeshot&hl=en

[20] Machines that can see: Convolutional Neural Networks. [Online; accessed 20.05.2018].
Retrieved from:
https://shafeentejani.github.io/2016-12-20/convolutional-neural-nets/

[21] Multilayer perceptron. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Multilayer_perceptron

[22] Mushroom Identify - Automatic picture recognition. Google Play. [Online; accessed 20.05.2018].
Retrieved from: https://play.google.com/store/apps/details?id=com.pingou.champignouf&hl=en

[23] Not Hotdog. App Store. [Online; accessed 20.05.2018].
Retrieved from:
https://itunes.apple.com/us/app/not-hotdog/id1212457521?mt=8

[24] Platform Architecture. Android Developers. [Online; accessed 20.05.2018].
Retrieved from: https://developer.android.com/guide/platform/

[25] Pokémon GO. Google Play. [Online; accessed 20.05.2018].
Retrieved from: https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo&hl=cs

[26] Precision-Recall. scikit-learn. [Online; accessed 20.05.2018].
Retrieved from: http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html

[27] Random forest. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Random_forest

[28] React. [Online; accessed 20.05.2018].
Retrieved from: https://reactjs.org

[29] React Native. [Online; accessed 20.05.2018].
Retrieved from: https://facebook.github.io/react-native/

[30] Redis. [Online; accessed 20.05.2018].
Retrieved from: https://redis.io

[31] Redux. [Online; accessed 20.05.2018].
Retrieved from: https://redux.js.org

[32] Supervised learning. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Supervised_learning

[33] Support vector machine. Wikipedia: the free encyclopedia. [Online; accessed 20.05.2018].
Retrieved from: https://en.wikipedia.org/wiki/Support_vector_machine

[34] TensorBoard: Visualizing Learning. TensorFlow. [Online; accessed 20.05.2018].
Retrieved from:
https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard

[35] Tensorflow Object Detection API. GitHub, Inc.. [Online; accessed 20.05.2018].
Retrieved from: https:
//github.com/tensorflow/models/tree/master/research/object_detection

[36] tiangolo/uwsgi-nginx-flask. Docker. [Online; accessed 20.05.2018].
Retrieved from: https://hub.docker.com/r/tiangolo/uwsgi-nginx-flask/

[37] VILOmeniny. [Online; accessed 20.05.2018].
Retrieved from: https://www.csfd.cz/film/251040-vilomeniny/prehled/

[38] Facebook: Facebook Login. facebook for developers. [Online; accessed 20.05.2018].
Retrieved from: https://developers.facebook.com/docs/facebook-login/

[39] Google: Emoji Scavenger Hunt. [Online; accessed 20.05.2018].
Retrieved from: https://emojiscavengerhunt.withgoogle.com

[40] Google: Google Sign-In. Google Identity Platform. [Online; accessed 20.05.2018].
Retrieved from: https://developers.google.com/identity/

[41] Google: TensorFlow.js. [Online; accessed 20.05.2018].
Retrieved from: https://js.tensorflow.org

[42] Krasin, I.; Duerig, T.; Alldrin, N.; et al.: OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*. 2017.

[43] Min-Ling ZHANG, X.-Y. L. X. G., Yu-Kun LI: Binary relevance for multi-label learning: an overview. *Front. Comput. Sci., 2018, 12(2)*. 2018: page 191–202. doi:https://doi.org/10.1007/s11704-017-7031-7.

[44] Ren, S.; He, K.; Girshick, R.; et al.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems 28*, edited by C. Cortes; N. D. Lawrence; D. D. Lee;

M. Sugiyama; R. Garnett. Curran Associates, Inc.. 2015. pp. 91–99.
Retrieved from: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf

[45] Russakovsky, O.; Deng, J.; Su, H.; et al.: ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*. vol. 115, no. 3. 2015: pp. 211–252. doi:10.1007/s11263-015-0816-y.

[46] Szegedy, C.; Vanhoucke, V.; Ioffe, S.; et al.: Rethinking the Inception Architecture for Computer Vision. 2015.

[47] Tsung-Yi Lin, S. B.-J. H. P. P.-D. R. P. D. C. L. Z., Michael Maire: Microsoft COCO: Common Objects in Context. *Dataset available from http://cocodataset.org/*. 2014.

[48] Twitter: Authentication. [Online; accessed 20.05.2018].
Retrieved from: https://developer.twitter.com/en/docs/basics/authentication/overview/oauth

[49] Wei Liu, D. E. C. S. S. R. C.-Y. F. A. C. B., Dragomir Anguelov: SSD: Single Shot MultiBox Detector. *ECCV 2016: Computer Vision – ECCV 2016*. 2016: pp. 21–37. doi:https://doi.org/10.1007/978-3-319-46448-0_2.

Appendices

List of Appendices

Appendix A

Average Precision per Object Class

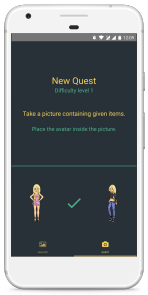
	Multi-label classification	Object detection	
object class	Inception V3 AP	Faster R-CNN AP	Single Shot Detector AP
bicycle	0.68	0.85	0.77
fire hydrant	0.81	0.86	0.84
cat	0.89	0.96	0.89
dog	0.87	0.94	0.85
backpack	0.57	0.56	0.60
tie	0.76	0.74	0.78
frisbee	0.82	0.90	0.84
skateboard	0.91	0.90	0.92
tennis racket	0.98	0.90	0.94
cup	0.61	0.79	0.75
fork	0.66	0.73	0.73
knife	0.48	0.49	0.60
spoon	0.39	0.49	0.49
banana	0.72	0.81	0.80
apple	0.51	0.59	0.58
orange	0.61	0.79	0.82
broccoli	0.78	0.89	0.87
pizza	0.83	0.92	0.91
donut	0.64	0.83	0.85
cake	0.70	0.82	0.81
chair	0.71	0.78	0.84
tv	0.82	0.88	0.89
laptop	0.77	0.87	0.85
cell phone	0.63	0.87	0.61
book	0.57	0.62	0.69
clock	0.76	0.84	0.78
scissors	0.44	0.68	0.60
teddy bear	0.69	0.90	0.80
toothbrush	0.53	0.66	0.63

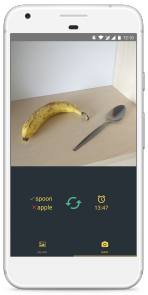
Appendix B

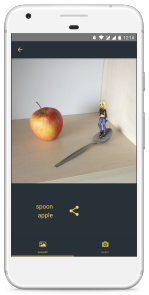
Presentation Poster

StageUp - A Photo-Challenge Mobile Application

Bc. Sebastián Poliak
Faculty of Information Technology, Brno University of Technology







Overview

- A mobile application, that challenges users to take a picture containing given items. Additional features such as animated avatars, time constrains on completing the quests or sharing the image on social media are added to the application, in order to promote playfulness and user interest. The application is designed to have a functionality, which automatically and instantly evaluates the taken pictures. This is addressed as an image recognition problem, and is solved using convolutional neural networks and multi-label classification.

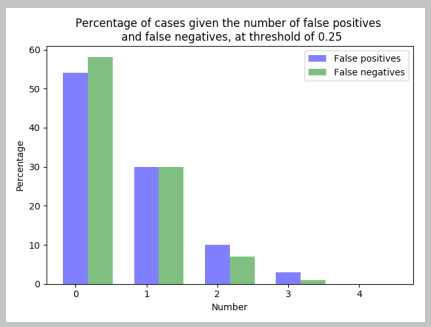
Technical design

- The architecture of the application is based on the client-server model. The client, in this case is the application running on a mobile device, is responsible mainly for taking a picture, communication with the server, and user interaction through the user interface, together with presentation of received data. The image recognition, as well as the functionality that generates and assigns quests to the users, are handled by the server. Another part of the server is the database, which makes the user profiles accessible from any device. The advantages and disadvantages of using this model have been taken into consideration.

Image recognition

- Two different approaches of image recognition have been implemented and considered. These are object detection and multi-label classification, trained on datasets such as The Open Images or COCO: Common Objects in Context. The models used for object detection are Faster R-CNN and Single Shot Detector (SSD), which ended up disregarded, mainly due to their high operation cost. The application ended up using Inception V3 model trained as a multi-label classifier, on 29 classes from COCO dataset.
- The actual usage of application will create a set of images, which can be used to further improve and evaluate the model. With the approval of the users, these images could be published for the community, as a new multi-label image dataset.

Evaluation



- A custom metric has been created, based on the number of false positives and false negatives per image in a multi-label setting. Considering the gameplay of the application, false positive represents the case when the application recognizes the object that is not in the picture, while false negative represents the case when the object is in the picture and does not get recognized. By increasing the threshold, the number of false positives decreases, while the number of false negatives increases. The threshold has been experimentally set to 0.25, where both false positives and false negatives are in the acceptable range. The metric has been evaluated using the multi-label classifier on the validation part of COCO dataset, taking into consideration only the 29 trained classes. Other metrics such as Precision-Recall have been evaluated as well, where the model reached mean average precision (mAP) of 0.69, over all classes.

Acknowledgments

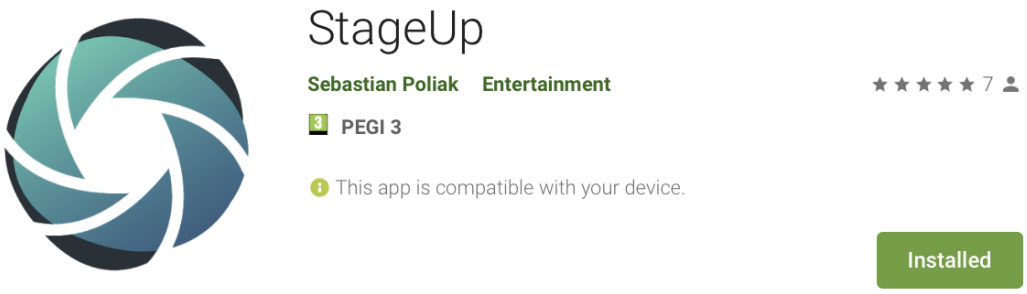
- The application was created as a diploma thesis, under the supervision of Ing. Jakub Sochor. Additional credits go to Ing, Rastislav Mrazik, the author of the animated avatars, and to Bc. Zdeno Olšovský.

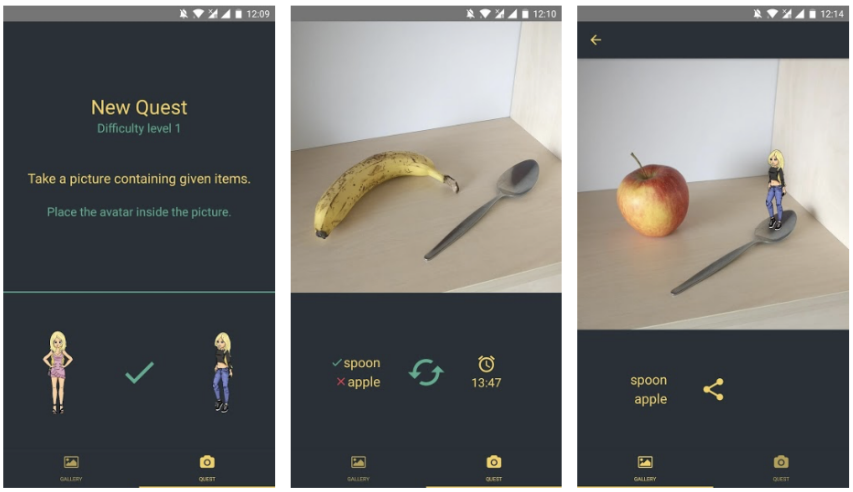
GET IT ON Google Play

xpolia01@stud.fit.vutbr.cz

Appendix C

Publication on Google Play





StageUp is a game, that challenges you to take a picture containing given items, within a certain time limit.
The difficulty of finding these items increases, as you level up and progress through of the game.
If you complete the quests, you can place your animated avatar inside the picture.
All your pictures are available in your gallery, and can be shared anytime.

Have fun and take some interesting pictures!

Appendix D

Content of Attached DVD

The attached DVD contains the following directories:

- `client` - source codes of the client side.
- `server` - source codes of the server side.
- `models` - the trained image recognition models, with the scripts that have been used for training and evaluation.
- `presentation` - the video and other presentational materials.
- `latex` - source codes of the thesis report

Appendix E

Manual

The easiest way to run the application, is to download and install it from https://play.google.com/store/apps/details?id=com.stageup.

Another way is to copy `client/app-release.apk` file into an Android device, and install it.

To build the application from the source code, it is needed to set up React Native according to https://facebook.github.io/react-native/docs/getting-started.html, and run `npm install` and `react-native run-android -variant=release`, from the `client/stageup` folder.